Escuela de Ingeniería de Sistemas y Computación

Facultad de Ingeniería

Sistema de Restricciones sobre Dominios Continuos para Gecode y Mozart

Trabajo de investigación para optar el título de Maestría en Ingeniería con énfasis en Ingeniería de Sistemas y Computación

Gonzalo José Hernández Garzón, Esp.

Director:

Juan Francisco Diaz Frias, Ph.D.

Codirector:

Gustavo Gutierrez,

Agosto, 2012

Resumen

El ejercicio profesional de un Ingeniero de Sistemas requiere solucionar todo tipo de problemas desde una mirada Sistemica y/o Computacional, encontrándose entre ellos un conjunto de problemas que requieren un tratamiento especial a nivel logarítmico por su complejidad computacional: *Problemas N-P*.

Los *problemas N-P*, que son problemas que no se pueden resolver con una complejidad polinomial, pueden ser resueltos con diferentes técnicas como: inteligencia artificial, algorítmos genéticos, programación por restricciones, entre otros.

La técnica de programación por restricciones propone un método de reducción de dominios en pro de acelerar la exploración de búsqueda de soluciones. Este proceso se lleva a cabo de manera sistemática y filtrando valores que descartan de forma rápida aquellos valores que no hacen parte de una solución.

Estos problemas se pueden modelar como restricciones sobre variables de diferentes tipos tales como enteros, booleanos, conjuntos y reales. Estos últimos que manejan dominios continuos, no han sido muy trabajados por los motores de búsqueda por restricciones actuales.

Gecode es uno de los motores de uso libre que permiten solucionar *problemas N-P* con programación por restricciones, pero hasta el momento no cuenta con un módulo que permita modelar problemas con variables de tipo real.

Mozart es un lenguaje de programación multiparadigma que actualmente esta trabajando en un proyecto que implementará las librerías de *Gecode* para su módulo de programación por restricciones.

Con esta investigación se propone un módulo que puede ser adicionado a *Gecode* de tal forma que permita ser utilizado en la solución de este tipo de problemas. De la misma forma, *Mozart* también podrá solucionarlos al momento de enlazarse con *Gecode*.

Agradecimientos

Conocer nuevas fronteras en la ciencia no solo enriquece el conocimiento propio, sino que a su vez crea nuevos caminos de exploración. El proceso de esta investigación, junto con el proceso mismo de la Maestría en Ingeniería de la Escuela de Ingeniería de Sistemas y Computación de la Universiad del Valle, aportó al autor, dos enormes campos de trabajo futuro:

En primer lugar, investigar sobre un campo nuevo solucionando problemas por medio de un paradigma que modela las situaciones utilizando restricciones; y en segundo lugar, profundizar en el uso de herramientas computacionales de software libre que en el transcurso de la investigación, permitieron avanzar de una forma más conveniente. Por ello agradezco de corazón a la Escuela y en especial al grupo de investigación AVISPA.

Los ingenieros que acompañaron este proceso investigativo fueron parte fundamental en el logro de los objetivos presentados en el Capítulo 5 de este documento. Juan Francisco, Gustavo Gutierrez, Gustavo Gomez y Andrés Farhat; mil gracias.

Dedicatoria

A mi querida esposa y a mis dos grandiosos hijos.

Índice

Índice de cuadros

Índice de figuras

1. Capítulo 1

1.1. Introducción

Los creadores de *Mozart*, han iniciado un nuevo proyecto denominado MVM con el ánimo de cubrir todos los posibles destinos que ha alcanzado este lenguaje [16]. El nuevo diseño contará con dos sistemas para interactuar, uno de Programación por Restricciones y otro para Programación Distribuida donde el Subsistema de Restricciones usará la librería de restricciones de *Gecode*.

Para el verano del año 2012 Christian Shulte (Desarrollador principal del proyecto), está pensado en la version 4.0.0 de *Gecode* que debería contener un módulo con el cual se puede establecer restricciones con dominios continuos.

GeOz es un proyecto desarrollado por el Grupo AVISPA con el cual se logró empalmar el motor de restricciones de *Gecode* a *Mozart*, con el fin de aprovechar propagadores más eficientes implementados en *Gecode*. Dicha implementación cuenta únicamente con los módulos necesarios para trabajar con Dominios Finitos (DF) y con Dominio de valores Booleanos (DB).

1.2. Problema

1.2.1. Definición

El proyecto *Máquina Virtual de Mozart (MVM)*, contará con las librerías de *Gecode* como nuevo motor de restricciones, pero *Gecode* no cuenta con el módulo necesario para trabajar restricciones sobre dominios continuos.

1.3. Objetivos

1.3.1. Objetivo General

Crear un sistema de restricciones sobre dominios continuos para *Gecode*, de tal forma que en el futuro puedan ser incluidos en el proyecto de la *Máquina Virtual de Mozart*.

1.3.2. Objetivos Específicos

- Implementar un módulo bajo *Gecode* para programar con restricciones sobre dominios continuos de tal forma que pueda dar solución a problemas que tengan que ver con números reales.
- Realizar las pruebas con los ejemplos recogidos como benchmarks.
- Analizar los resultados obtenidos después de realizar las pruebas con los benchmarks.

1.4. Estructura del documento

Este documento es un informe que redacta todos los momentos de la investigación realizada para lograr obtener un módulo apropiado que trabaje sobre variables de dominios continuos para *Gecode* y *Mozart*.

El **Capítulo 1** presentó la Introducción, la Descripción del problema y los Objetivos a cumplirse al final del proceso. El **Capítulo 2** enmarca algunos conceptos necesarios para comprender y fundamentar la investigación. El **Capítulo 3** describe la implementación de *CPFloat-Gecode* que solventa el problema definido en el primer capítulo. Los **Capítulo 4 y 5** dan informe de las pruebas realizadas para verificar el eficiencia de *CPFloat-Gecode*. Finalmente, el **Capítulo 6** expone las conclusiones obtenidas al finalizar el proceso investigativo, asi como los trabajos que se pueden realizar en el futuro para mejorar a *CPFloat-Gecode*.

2. Capítulo 2

2.1. Marco Teórico

2.1.1. Matemática Intervalar con Reales [2][3][4]

Expresar un número real de esta forma es relativamente fácil a nivel de computación, sin embargo no es lo único que se debe soportar. Hay que tener en cuenta que los intervalos son operados con otros números por medio de sumas, restas, multiplicaciones, divisiones, e.t.c., generando nuevos intervalos. Por esta razón también hay que soportar operaciones entre ellos.

Si queremos extender las operaciones entre reales tales como suma, resta, multiplicación y división a los intervalos, es necesario pensar en que dicha operación también debe ser operada entre cada uno de los elementos de los mismos. Es decir:

Si tenemos: $\forall p \subseteq \mathbb{R}^n, \square(p) = \cap\{B \in \mathbb{I}^n | p \subseteq B\}$
entonces para $\diamond \in \{+, -, \times, \div\}$ tenemos que $X \diamondsuit Y = \square\{x \diamond y | x \in X, y \in Y\}$

Aun si la operación está sujeta a cada uno de los elementos del intervalo, la ventaja de trabajar con éstos radica en que solo basta con calcular las operaciones con los extremos de cada uno y se mantendrá la veracidad en el resultado.

Las operaciones con *Intervalos de Reales* son similares a las de los *Intervalos con Flotantes*, se diferencian por la necesaria utilización de la función techo $\lceil x \rceil$ y la función piso$\lfloor x \rfloor$ para aproximar un valor real por arriba o por debajo respectivamente:

$\lceil x \rceil = min\{k \in \mathbb{F} | x \leq k\}$
$\lfloor x \rfloor = max\{k \in \mathbb{F} | k \leq x\}$

De esta forma se puede operar intervalos flotantes como se muestra a continuacion:

$[a, b] + [c, d] = [\lfloor a + c \rfloor, \lceil b + d \rceil]$
$[a, b] - [c, d] = [\lfloor a - c \rfloor, \lceil b - d \rceil]$
$[a, b] \times [c, d] = [min(\lfloor ac \rfloor, \lfloor ad \rfloor, \lfloor bc \rfloor, \lfloor bd \rfloor), max(\lceil ac \rceil, \lceil ad \rceil, \lceil bc \rceil, \lceil bd \rceil)]$

Con lo anterior se puede deducir como ejemplo que la operación para el cálculo de la longitud de una circunferencia de radio 3, se podrá calcular como:

$l = \pi \cdot (2 \cdot r)$
$l = \pi \cdot (2 \cdot 3)$
$l = [3{,}141592653589793238\ ,\ 3{,}141592653589793239] \times ([2, 2] \times [3, 3])$
$l = [3{,}141592653589793238\ ,\ 3{,}141592653589793239] \times [6, 6]$

En el proceso de cálculo de intervalos es necesario entonces tener en cuenta ciertas propiedades las cuales se cumplen todas para intervalos de reales y no todas para intervalos de flotantes.

- Conmutativa:
  $X + Y = Y + X$ o
  $X \cdot Y = Y \cdot X$

- Asociativa:
  $(X + Y) + Z = X + (Y + Z)$ o
  $(X \cdot Y) \cdot Z = X \cdot (Y \cdot Z)$
- Sub-distributiva:
  $X \cdot (Y + Z) \subseteq X \cdot Y + X \cdot Z$

La propiedad asociativa solo se cumple para intervalos con reales puesto que para intervalos con flotantes, el uso de las funciones de piso y techo cambian el resultado.

Además de tener en cuenta estas operaciones, se requiere que el resultado esté representado por un intervalo que sea lo suficientemente angosto como para que cualquiera de sus elementos sean válidos como solución a un problema. A medida que las sub-operaciones se van ejecutando, el intervalo resultante tiende a aumentar su tamaño.

Dada una ecuación donde se involucran intervalos, es posible angostar el rango del intervalo resultante. Por ejemplo, para una funcion dada:

Sobre reales:	$f(x)$	$= x(x + 1)$
Sobre intervalos:	$F(X)$	$= X(X + [1, 1])$

Se pretende evaluar la función para $x \in [-1, 1]$. Es posible transformar su predicado para que el intervalo resultante sea más angosto. A continuacion, 4 diferentes opciones para escoger:

$$
\begin{aligned}
F_1(X) &= X(X + 1) && = [-2, 2] \\
F_2(X) &= XX + X && = [-2, 2] \\
F_3(X) &= X^2 + X && = [-1, 2] \\
F_4(X) &= (X + \tfrac{1}{2})^2 - \tfrac{1}{4} && = [-\tfrac{1}{4}, 2]
\end{aligned}
$$

Como se puede notar, F_4 genera el intervalo más angosto. Esto se debe a que en ella es donde aparece una sola ocurrencia de X.

Teorema: [12] Sea $f : \mathbb{R}^n \to \mathbb{R}$ una función real tal que ninguna variable ocurre más de una vez, y $F : \mathbb{I}^n \to \mathbb{I}$ su extensión intervalar natural de f; entonces:

$F(X_1, ..., X_n) = \{f(x_1, ..., x_n) | x_i \in X_i, i = 1, ..., n\}$

2.1.2. Nociones de Consistencia

A nivel global todos los algorítmos utilizados para resolver problemas por medio de programación por restricciones, se fundamentan en alguna de las técnicas exisitentes de *Consistencia*. Estas técnicas ya han sido revisadas y aceptadas por la comunidad científica.

Inicialmente, se establece que un Problema de Satisfacción por Restricciones CSP (Constraint Satisfaction Problem), es consistente, si todos y cada uno de los valores que hacen parte del dominio son solución del CSP. Para tal fin el sistema de restricciones deberá proveer los algoritmos necesarios para lograr eliminar de dicho dominio todos los elementos con los cuales alguna de las restricciones no satisface el problema.

Este tipo de *Consistencia Total*, no es fácil de lograr en este proyecto debido a que los valores a trabajar se entienden como continuos. Los valores reales por su naturaleza, dificultan la evaluación de cada uno de los mismos y solo se puede contar con algunos modelos matemáticos que aseguran su veracidad pero no son aplicables a todos los casos.

En general, muchos de los algoritmos utilizados se basan en *Técnicas de filtro de Consistencia Parcial* [4]. A continuación, se establecerá los aspectos fundamentales de estas técnicas, las cuales han sido utilizadas en el trabajo de restricciones sobre dominios continuos.

2.1.3. Limite Consistencia

Si bien la fuerza de filtro que se puede lograr con una *Técnica de Fitro de Consistencia Total*, es ideal, su implementación podría requerir de un trabajo local dispendioso.

La técnica *Limite Consistencia* (*Bound-Consistency*), relaja la intensidad del filtro evaluando la consistencia solo en los límites del dominio. Para la mayoría de los casos, esta evaluación es suficiente además de ser económica para el procesamiento.

2.2. Estado del Arte

2.2.1. Algoritmos que trabajan con números reales

En todo algoritmo se requiere hacer operaciones matemáticas; aun si solo se pretende poner un texto en la pantalla, se necesitará calcular trigonométricamente su posición. Estos cálculos son realizados de forma transparente para el usuario y en ocasiones, también trasparentes para el programador.

Si es verdad que la mayoría de cálculos se realizan con números enteros, un gran porcentaje de programas requieren cálculos con números reales, en especial, aquellas aplicaciones que simulan eventos físicos. Para llevar estos algoritmos de cálculos que operan con números reales, a un computador, es necesario hacerlo por medio de tipos de datos de *Punto Flotante (floating point numbers)*. Tipos que con un número limitado de decimales, se aproximan a un valor real.

Un tipo de dato de *Punto Flotante* según el último estandar IEEE 754 [5], es un número finito ya sea en base binario (en base 2) o decimal (en base 10), expresado por 3 números enteros que definen: el signo, un coeficiente y un exponente.

$(-1)^s \cdot c \cdot b^q$

Donde s=signo, c=coeficiente, q=exponente, b=base(2 o 10)

En el estandar también se definen formatos de intercambio, operaciones, métodos de redondeo y manipulación de excepciones.

En un ejemplo clásico, si se requiere calcular la longitud de una circunferencia, será indispensable en el proceso tener el valor para π en:

$l = \pi \cdot 2r$

En un programa, este valor se almacena en una variable de *Punto Flotante* con el valor de: 3.1415 9265 3589 7932 3846 y con el doble de capacidad (128 bits) en una variable de *Punto Flotante largo* con el valor de: 3.1415 9265 3589 7932 3846 2643 3832 7950 29. Por supuesto, estos valores se acercan al valor real de π, pero no es el valor de π verdadero [13].

El valor aproximado para π según la definicion de la IEEE 754 [5], se almacenará en un tipo de dato de punto flotante asi:

$(-1)^0 \cdot 314159265358979323846 \cdot 10^{-20}$

Para muchos algoritmos este valor es suficiente puesto que el resultado satisface una solución, pero en el resto de programas este truncamiento de los decimales puede generar errores extremos.

Con computaciones de este tipo es posible encontrar tres tipos de errores:

- Redondeo: El valor flotante no representa un número real, tal como se vió en el caso del número π.
- Absorción: Inexactitud de los resultados debido a que los valores almacenados son muy pequeños y se pierden al ser operados con otros valores más grandes.
- Cancelación: Inexactitud en los resultados debido a redondeos parciales en suboperaciones. [7]

Desde el punto de vista matemático, la aritmética intervalar surge como solución a los inconvenientes descritos anteriormente, manteniendo la correctitud de los resultados al ser expresados en intervalos.

De esta forma, un número real inexpresable por su infinidad de decimales, puede ser presentado como un intervalo de reales.

$a, b \in \mathbb{R} : [a, b] = \{x \in \mathbb{R} \mid a \leq x \leq b\}$

En este caso, el valor de π puede ser visto como un intervalo así:

$\pi =$ [3,14159265358979323846 , 3,14159265358979323847]

La anterior expresión significa que el valor real para π se encuentra entre el número 3.1415 9265 3589 7932 3846 y el número 3.1415 9265 3589 7932 3847.

Este mecanismo es útil a la hora de ser implementado en un computador puesto que el Intervalo de *Números Reales* puede ser acondicionado a *Intervalo de Numeros Flotantes*, que igual representarían a un número real. En ese evento la formalidad sería reemplazada por:

$a, b \in \mathbb{F} : [a, b] = \{x \in \mathbb{R} \mid a \leq x \leq b\}$

Aun así, π seguiría teniendo el mismo valor intervalarmente.

Esta formalidad matemática da certeza y a la vez practicidad a la hora de escoger un resultado pudiendo tomar el valor inferior o el superior o uno entre ellos dos. El éxito de esta técnica está en cerrar el intervalo [3] lo suficiente como para que el resultado sea suficientemente próximo al resultado exacto manteniendo un costo computacional razonable. Esto se tratará posteriormente.

2.2.2. Lenguajes de Programación por Restricciones

Entre los diversos lenguajes de programación que existen, hay unos dedicados a desarrollar programas bajo el paradigma de Programación por Restricciones [1][2], modelo declarativo de alto nivel que busca por medio de modelos matemáticos, formular restricciones que hacen que de un dominio de soluciones, solo se tomen aquellas que satisfagan una solución a un problema. Por ejemplo, si el problema consiste en encontrar los números primos que se encuentren entre el rango de 1 a 100, el CSP por medio de restricciones, excluirá a todos aquellos que no cumplan con la característica de ser únicamente divisibles entre ellos mismos y la unidad.

Este trabajo de investigación encierra su estudio a 3 lenguajes de CSP especificos, *Mozart*, *Gecode* y *GeOz*.

Mozart [14]
Es un lenguaje de programación multiparadigma bastante expresivo, que permite codificar algorítmos de una forma cómoda. Debido a que tiene implementado un módulo para programación por restricciones, es tomado como objetivo de esta investigación.

Gecode [15]
Es un ambiente de desarrollo basado en restricciones implementado en lenguaje C++ con atractivas características: Código abierto, Libre, Portable y Accesible. *Gecode* nace experimentando el módulo de restricciones para *Mozart*, para luego reprogramar todo su código en uno nuevo denominado *Gecode*, más eficiente y vigente.

GeOz [18]
No es un lenguaje propiamente dicho, sino un proyecto que busca tomar todas las ventajas de desempeño de *Gecode* y acoplarlas como nuevo módulo de restricciones en *Mozart*.

2.2.3. Trabajos Relacionados

Implementación Aritmética Intervalar de Hickey
Con la documentación escrita por Hickey [10] sobre aritmética intervalar, se ha creado una librería bajo lenguaje de programación C++ que implementa dicha teoría; su fuente se encuentra disponible para usar en Bosts.org [17]. Esta librería funciona con tipos de datos de punto flotante aunque no garantiza su correcto funcionamiento en todos los sistemas de cómputo debido a la heterogeneidad de sus arquitecturas. Esta librería es una colección de clases las cuales pueden ser usadas para trabajar con intervalos variando sus políticas de desempeño.

Variables Flotantes para Gecode
Como un trabajo adicional, Filip Konvicka, implementó algunos propagadores sobre reales usando intervalos, como una necesidad específica de su trabajo de investigación. Estos propagadores fueron creados bajo el código de *Gecode*.

Sistema de Restricciones de Aritmética Intervalar XRI
Es un sistema de restricciones sobre números reales implementado para el lenguaje de programación *Mozart* [19]. Además cuenta con una interface para mezclar los cómputos de reales con los solvers de programación lineal de números enteros previamente implementados. Su nombre viene de el

acrónimo eXtended Real Interval.

3. Capítulo 3

El sistema de clases de *Gecode* está previsto para ser extendido de forma sistemática, de tal forma que un usuario programador puede extender fácilmente nuevos *espacios de variables*, nuevos *propagadores* y también nuevos *distribuidores* para lograr una buena personalización del código. Con este proyecto se logró extender un nuevo *tipo de dominio de variables* haciéndose necesaria la recompilación de *Gecode*.

Para que *Gecode* pueda ofrecer *Programación por Restricciones* sobre Dominios Continuos, se ha diseñado e implementado un módulo escrito en lenguaje c++ conservando la filosofía del mismo. A este módulo se le ha denominado *CPFloat-Gecode* (Constraint Programming Float for Gecode). En las siguientes cuatro secciones (CPFloat, Propagadores basicos/avanzados y Distribuidores) se detalla su diseño e implementación.

3.1. CPFloat

Tal como se muestra en la Figura 1, CPFloat aprovecha la posibilidad de extensión de *Gecode* agregándole 5 clases principales, a saber: CPFloatVarImp, CPFloatVar, CPFloatView, CPFloatVarArray y CPFloatVarArgs. Estas clases heredan características de las clases VarImpVar y VarImpView y VarArray del sistema de *Gecode* diseñadas precisamente para este propósito.

La funcionalidad del sistema característico de *Gecode* está enmarcada por un modelo multi-capa que busca responsabilizar a cada nivel de tareas diferentes, por tal motivo es necesaria la creación de todas estas clases.

Las 5 clases definidas se han agrupado dentro un espacio reservado denominado MPG (corresponde al área de color rosado), entendiéndose que todo lo que se encuentra dentro de este espacio hace parte del módulo *CPFloat-Gecode*. Internamente se ha clasificado algunas de estas clases en un subespacio denominado CPFloat (área de color azul). Esta separación está predefinida en estándares propuestos por *Gecode* para futuros desarrollos.

A continuación se detalla las 5 clases principales.

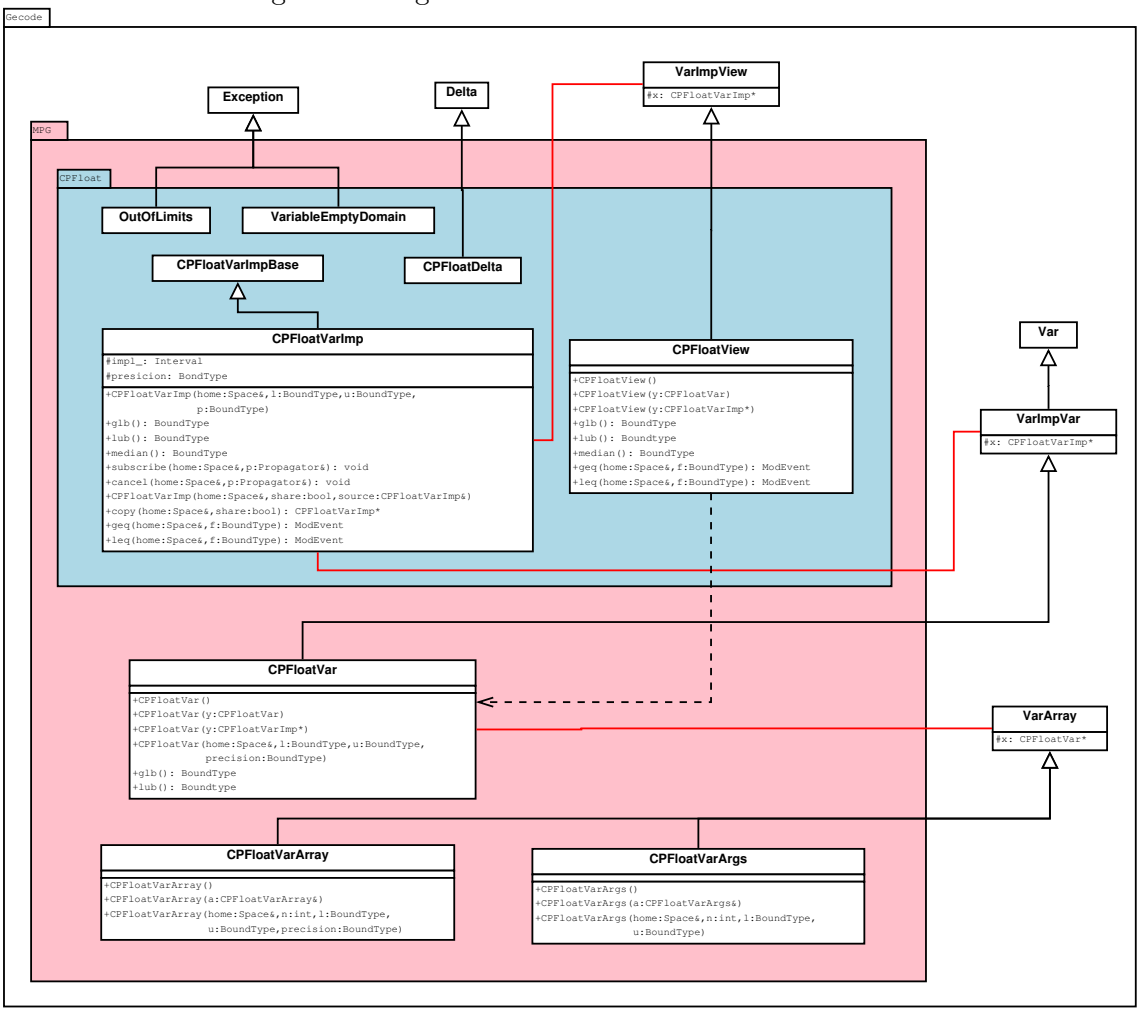
3.1.1. CPFloatVarImp

Esta clase es la que está más cerca del kernel de *Gecode*, hereda todas sus características de la clase CPFloatVarImpBase, la cual fue creada directamente en el núcleo del sistema cuando fue compilado en el proceso de Instalación de *CPFloat-Gecode*.

CPFloatVarImp, se encarga de administrar el valor de la variable real, en este caso el valor será almacenado en un intervalo limitado por dos números flotantes. Para su apropiado manejo se ha reutilizado el paquete *smath* creado por Tim Hickey [1]. Este paquete de archivos codificados en "C ANSI" hace un apropiado manejo de valores flotantes en un intervalo.

En el Cuadro 1 se presentan los 5 servicios (métodos) proveídos por esta clase: 3 para consulta de valores y 2 para modificación.

Figura 1: Diagrama del núcleo de *CPFloat-Gecode*



Cuadro 1: Métodos de la clase *CPFloatVarImp*

`glb() //greatest lower bound`	Retorna el valor mas bajo del intervalo
`lub() //least upper bound`	Retorna el valor mas alto del intervalo
`median() //madian value`	Retorna el valor medio del intervalo (glb+lub) / 2
`geq() //great or equal to ...`	Modifica el intervalo para que sea mayor o igual a un valor dado
`leq() //least or equal to ...`	Modifica el intervalo para que sea menor o igual a un valor dado

Los demás métodos son aquellos requeridos por *Gecode* para que los valores puedan ser manipulados como “un todo” en un mismo “Espacio”.

3.1.2. CPFloatVar

Esta clase se encuentra en la capa inmediatamente superior a CPFloatVarImp, es la clase que utiliza el usuario programador para programar sus restricciones.

CPFloatVar hereda de la clase VarImpVar del sistema de *Gecode*, de la misma clase de la cual heredan los otros tipos de variables de *Gecode*. Otro ejemplo de tipo variable disponible ya predefinida es IntVar utilizada para trabajar con dominios de enteros.

Cuando en un ejercicio se instancia un CPFloatVar, internamente se asocia una variable CPFloatImpVar identificada con x (A través de la herencia que se hace de VarImpVar). Esta variable x es la puerta para llegar al valor del numero real que se requiere representar.

En esta capa solo se dispone principalmente de los servicios de consulta de valores, tal como se muestran en el Cuadro 2.

Cuadro 2: Métodos de la clase *CPFloatVar*

`glb() //greatest lower bound`	Retorna el valor mas bajo del intervalo
`lub() //least upper bound`	Retorna el valor mas alto del intervalo

3.1.3. CPFloatView

La clase CPFloatView está en una capa que proporciona una vista de los datos, para tal fin usa a la clase CPFloatVar para extraer una referencia de la variable de implementación x de tipo CPFloatVarImp.

Esta clase también dispone de 3 servicios para consulta y 2 para modificación, los cuales están implementados para que hagan lo propio en el nivel inferior (CPFloatVarImp). El Cuadro 3 detalla los métodos.

En esta capa solo se dispone principalmente de los servicios de consulta de valores, tal como se muestran en el Cuadro 3.

Cuadro 3: Métodos de la clase *CPFloatView*

`glb() //greatest lower bound`	Retorna el valor mas bajo del intervalo
`lub() //least upper bound`	Retorna el valor mas alto del intervalo
`median() //madian value`	Retorna el valor medio del intervalo (glb+lub) / 2
`geq() //great or equal to ...`	Modifica el intervalo para que sea mayor o igual a un valor dado
`leq() //least or equal to ...`	Modifica el intervalo para que sea menor o igual a un valor dado

Con esta clase, el motor de búsqueda podrá consultar los valores del dominio y a la vez podrá también modificarlos a través de los propagadores.

3.1.4. CPFloatVarArray / CPFloatVarArgs

Estas dos clases se comportan como una interface y están a la altura de la capa en la que se encuentra CPFloatVar. Por medio de estas clases, el usuario programador podrá definir varias variables de tipo CPFloatVar arregladas en una sola colección.

Esta capacidad de trabajar n variables de forma transparente es lograda gracias a la herencia obtenida de la clase VarArray del sistema *Gecode*. Los métodos codificados en esta clase son los requeridos para que trabaje adecuadamente con el resto del sistema, los demás métodos son heredados. Es importante detallar uno de los métodos heredados:

Cuadro 4: Método heredado de la clase *VarArray*

```
CPFloatVar& operator [](int i); // array access method
```

El método descrito en el Cuadro 4 hace posible extraer del arreglo cualquiera de las variables de acuerdo con la posición deseada de forma simple usando los corchetes cuadrados.

La diferencia esencial entre las dos clases radica en que CPFloatVarArray colecciona variables para los dominios, en cambio CPFloatVarArgs es utilizada para definir variables auxiliares.

3.2. Propagadores Básicos

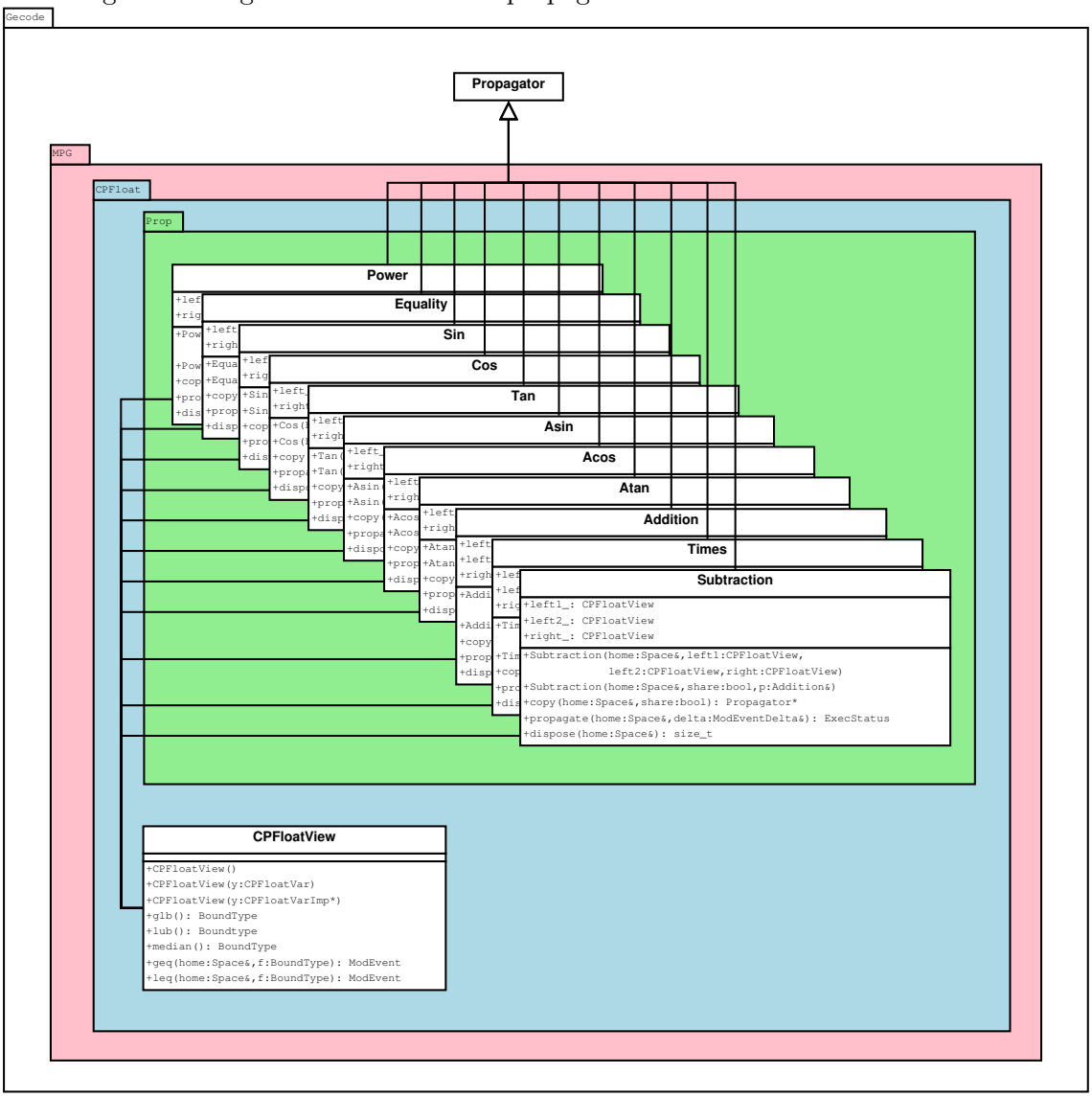
Como se revisó en las anteriores secciones, *CPFloat-Gecode* está diseñado para administrar dominios de valores continuos. En este capítulo se detallan las clases adicionales creadas para ofrecer al usuario programador la posibilidad de implementar restricciones sobre variables reales.

Con el fin de poder establecer restricciones sobre las variables reales, se creó un nuevo espacio denominado Prop, tal como se presenta en la Figura 2 (área de color verde), dentro del cual se crearon 11 nuevas clases: 4 para implementar restricciones aritméticas, 6 para implementar restricciones trigonométricas y 1 para implementar restricciones de igualdad.

CPFloat-Gecode solo provee Límite Consistencia (Bound-Consistency) [1], es decir, solo se hace análisis de consistencia en los límites del intervalo que representa a la variable real, de tal forma que los dominios de cada variable son podados únicamente en los extremos. [2]

A continuación se analizan los propagadores clasificándolos en 3 grupos: Aritméticos, Trigonométricos y de Igualdad.

Figura 2: Diagrama del sistema de propagadores básicos de *CPFloat-Gecode*



3.2.1. Propagadores Aritméticos

Las clases Addition, Subtraction, Times y Power, están diseñadas para implementar las cuatro operaciones básicas sobre las variables reales.

Addition, Subtraction y Times pueden ser instanciadas por el usuario programador sobre 3 CP-FloatVar's. Ejemplo:

```
CPFloatVar a(*this, -3.6, 8.4);
CPFloatVar b(*this, 1.8, 12.9);
CPFloatVar C(*this, 2.7, 6.3);
```

```
times(*this, a, b, c);
```

En este caso se está imponiendo una restricción de multiplicación, de tal forma que las 3 variables cumplan la condición de que:

$a * b = c$

Se aclara que el método times es un método que no es miembro de clase, el cual internamente invoca al constructor de la clase Times.

Para este propósito, estas clases tienen asociado 3 vistas (CPFloatView) que proveen el vínculo con las variables reales, estas vistas se describen en el Cuadro 5.

Cuadro 5: Atributos para propagadores aritméticos trinarios

`left1_ //first left view before to equal`	Vista que permite accesar a la primera variable
`left2_ //second left view before to equal`	Vista que permite accesar a la segunda variable
`right_ //right view after to equal`	Vista que permite accesar a la tercera variable

Por otro lado, la clase Power permite establecer restricciones de exponenciación de una variable CPFloatVar exponenciada a una constante entera.

Teniendo en cuenta que el exponente no es un CPFloatVar, el Cuadro 6 muestra que esta clase internamente solo está provista de 2 vistas:

Cuadro 6: Atributos para propagador aritmético binario

`left_ //view before to equal`	Vista que permite accesar a la primera variable
`right_ //view after to equal`	Vista que permite accesar a la segunda variable

Ejemplo:

```
CPFloatVar a(*this, 0.6, 4.5);
CPFloatVar b(*this, -4.8, 13.8);


power(*this, a, 2, b);
```

En esa ocasión, la restricción propuesta impone una restricción que busca encontrar valores en a y en b que satisfagan la condición de que:

$a^2 = b$

3.2.2. Propagadores Trigonométricos

CPFloat-Gecode también dispone de 6 propagadores que proporcionan la funcionalidad de imponer restricciones de tipo seno, coseno,tangente, sobre variables reales. Ejemplo:

```
CPFloatVar a(*this, 2.2, 4.6);
CPFloatVar b(*this, 0.0, 7.7);


sin(*this, a, b);
```

La operación anterior impone una restricción sobre los dominios de a y b que buscan aquellos valores que cumplan con la condición de que:

$sin(a) = b$

Las clases provistas de este tipo son: Sin, Cos, Tan, Asin, Acos y Atan; cada una de ellas internamente está asociada a 2 vistas, tal como se muestra en el Cuadro 7

Cuadro 7: Atributos para propagadores trigonométricos binarios

`left_ //view before to equal`	Vista que permite accesar a la primera variable
`right_ //view after to equal`	Vista que permite accesar a la segunda variable

3.2.3. Propagador de Igualdad

El sistema de restricciones para dominios continuos *CPFloat-Gecode*, provee la clase Equality que logra imponer una restricción como lo muestra el siguiente ejemplo:

```
CPFloatVar a(*this, -3.4, 7.1);
CPFloatVar b(*this, -9.5, 4.0);


equality(*this, a, b);
```

En este ejemplo se muestra cómo es posible buscar una intersección entre el dominio de a y el dominio de b, es decir, que este propagador busca: cuáles valores de a y b satisfacen la condición

de que:

$a = b$

Al igual que los propagadores trigonométricos, esta clase Equality contiene dos vistas que se muestran en el Cuadro 8

Cuadro 8: Atributos para propagador de igualdad binario

`left_ //view before to equal`	Vista que permite accesar a la primera variable
`right_ //view after to equal`	Vista que permite accesar a la segunda variable

3.3. Propagadores Avanzados

En esta sección se ha clasificado la descripción de dos propagadores codificados con algoritmos especiales. Estos propagadores están diseñados para implementar restricciones que involucran "n" variables dispuestas en una expresión matemática.

El primero es HC4 que implementa una versión de 2B-Consistencia, el segundo es K3B el cual se fundamenta en la técnica de filtrado de 3B-Consistencia.

3.3.1. Propagador HC4

Este propagador está codificado de forma tal que implementa la técnica de consistencia "Hull-Consistency"[4] sobre una restricción de características similares a las mostradas en los siguientes ejemplos:

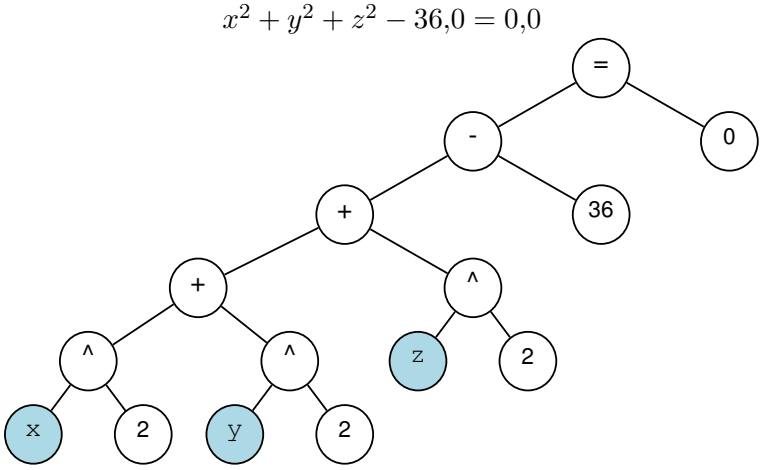
$x^2 + y^2 + z^2 - 36{,}0 = 0{,}0$ *(Primera expresión de Bronstein)*
$y^2 * z + x * y * t * 2{,}0 - x * 2{,}0 - z = 0{,}0$ *(Primera expresión de Caprasse)*

Para tal fin HC4 elabora un árbol binario para organizar cada una de las expresiones matemáticas y asi subdividir el problema en pequeñas operaciones de máximo dos variables [3], proporcionando arco-consistencia parcial. Un ejemplo de la estructura creada por HC4 es mostrada en la Figura 3.

Para implementar el algoritmo, el propagador HC4 no dispone de vistas de las variables en forma directa, en su lugar tiene la raiz de un árbol binario en el cual sus nodos contienen las operaciones matemáticas y en las hojas se sitúan las vistas de las variables reales y tambien los valores constantes.

Internamente el propagador HC4 trabaja recursivamente sobre el árbol lógico creado. La Figura 4 muestra el esquema general de su estructura. La clase HC4 tiene como atributo una referencia a un objeto de tipo Constraint la cual relaciona por igualdad a una expresión por izquierda y otra expresión por derecha.

Figura 3: Ejemplo de árbol creado por *HC4*



Cada expresión corresponde a una referencia de un objeto de tipo *Expression*, el cual contiene recursivamente a otras expresiones matemáticas.

En el diagrama mostrado por la Figura 4 se visualizan 6 clases que permiten administrar el árbol lógico: *Constraint*, *Expression*, *VarExpression*, *ConstExpression*, *UnaryExpression* y *BinaryExpression*.

La familia de clases *Expression* proporciona la flexibilidad para representar cualquier expresión matemática que contenga las operaciones implementadas en esta primera versión: *+, -,*, /, pow, sin, cos, tan, asin, acos, atan.*

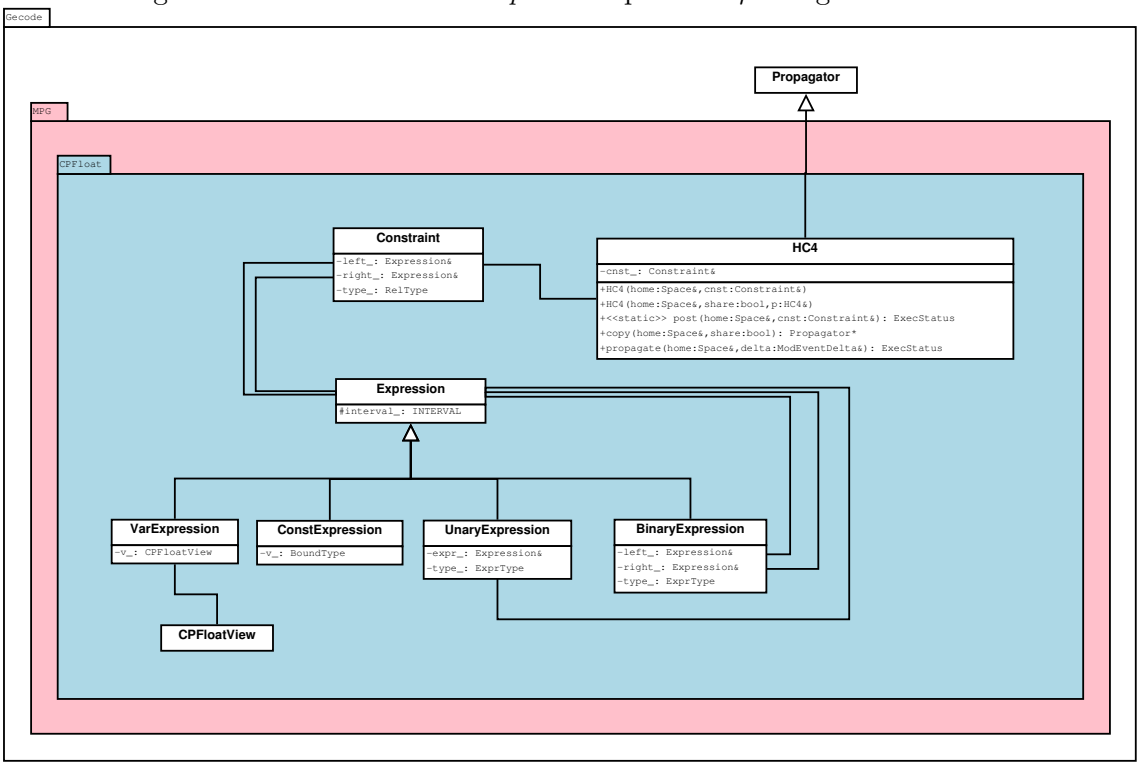
La Figura 5 muestra en forma detallada los miembros de cada una de las clases esquematizadas en la Figura 4 y los siguientes apartes explican a cada una de ellas en forma independiente.

Clase Expression *Expression* es la clase bases no-instanciable de la cual se desprenden todas las demás. Gracias al polimorfismo, sus métodos virtuales hacen visibles todas las operaciones entre los elementos de la expresión. En el Cuadro 9 se describen los miembros de la clase *Expression* que se invocan en forma recursiva alcanzado a todas las vistas de variables flotantes ubicadas en las hojas del árbol.

En el Cuadro 10 se describe los métodos de la clase *Expression* que no serán sobrecargados por las subclases. De estos métodos se destacan los que sobrecargan métodos de tipo operador, ya que son utilizados para armar la expresión matemática de forma natural. De esta forma el usuario programador podrá fácilmente escribir una ecuación y enviarla como parámetro al propagador.

Clase VarExpression *VarExpression* es la clase que instancia los nodos finales de las ramas del árbol, y es el lugar donde se encuentran las Vistas a las variables CPFloatVar. Es en esta clase donde la recursividad de los métodos sobrecargados inicia el desapilamiento.

Figura 4: Familia de clases *Expression* para *HC4*- Diagrama General



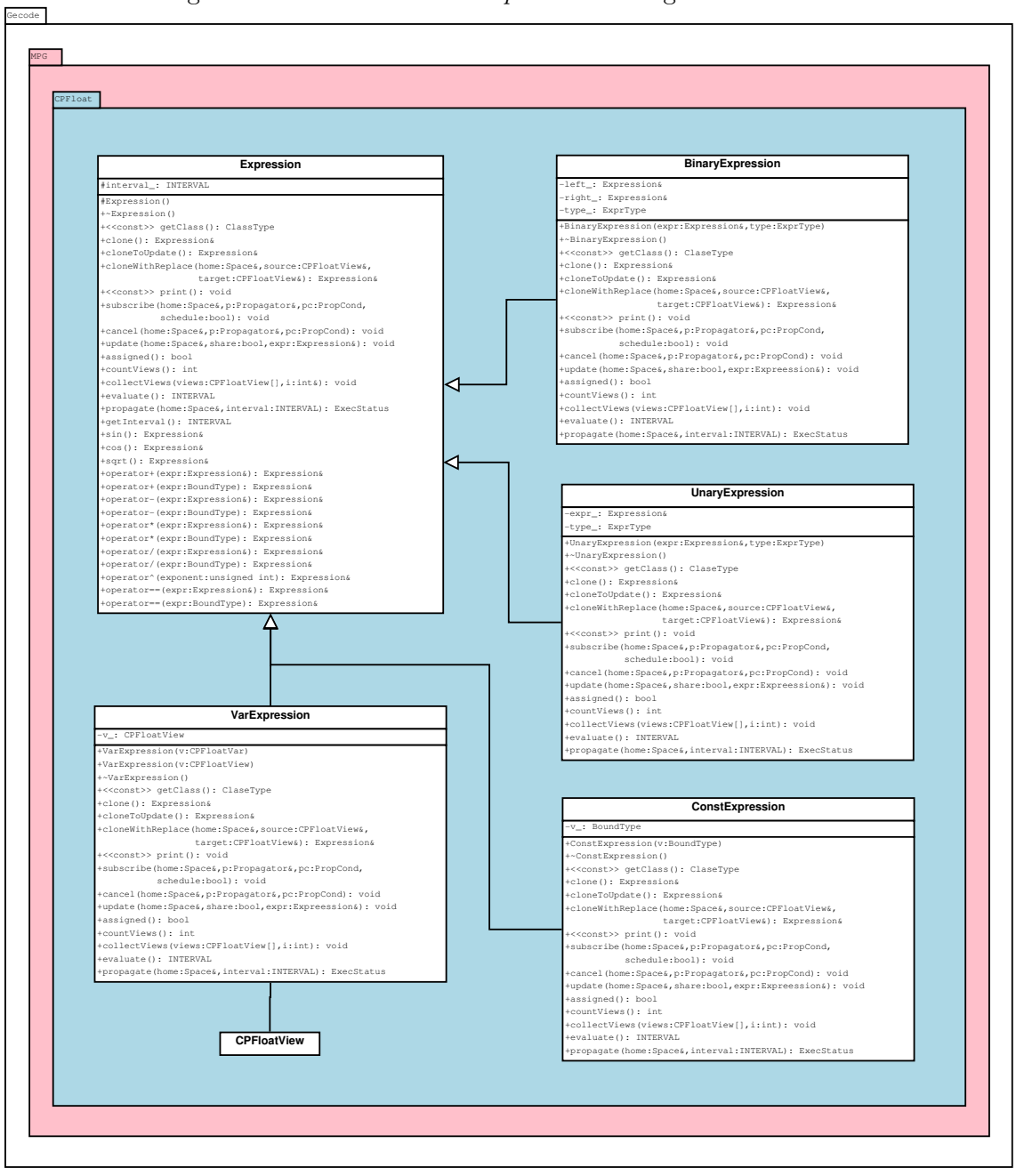
El Cuadro 11 detalla los métodos constructores, los miembros restantes son listados pero no detallados debido a que son sobrecarga de los métodos heredados de la clase *Expression*.

En este nivel del arbol (Final de las ramas), los métodos sobrecargados ejecutan los procedimientos directamente sobre las Vistas coleccionadas por el Propagador. El Cuadro 12 lista los métodos detallando su funcionalidad.

Clase ConstExpression *ConstExpression* esta diseñada para poder trabajar con valores constantes dentro de la expresión matemática. Al igual que *VarExpression*, esta clase instancia nodos al final de las ramas del árbol. En el Cuadro 13 se describen los métodos constructor y destructor. Y en el Cuadro 14 se detallan algunos apartes de cómo esta clase sobrecarga los métodos recursivos del sistema.

Clase UnaryExpression *UnaryExpression* instancia objetos en los puntos intermedios del árbol, operando en forma única a un objeto de tipo *VarExpression* o a uno de tipo *ConstExpression*. Esta clase implementa las operaciones unarias como *sqrt* y las trigonométricas tales como *sin, cos*. El Cuadro 15 detalla el Constructor y destructor de la clase. El Cuadro 16 describe cómo esta clase sobrecarga los métodos recursivos.

Figura 5: Familia de clases *Expression* - Diagrama detallado



Clase BinaryExpression *BinaryExpression* crea objetos que operan a dos expresiones, estas pueden ser de diferente tipo: $+, -, *, /$ o $\hat{}$. El Cuadro 18 lista y describe el funcionamiento de las funciones recursivas al momento de ser sobrecargadas.

Cuadro 9: Descripción de la clase *Expression* - Métodos recursivos

`ClassType getClass()`	Método que averigua a que tipo corresponde el objeto instanciado.
`Expression& clone()`	Método utilizado para clonar el arbol en forma recursiva
`Expression& cloneToUpdate()`	Método utilizado para clonar el arbol en forma recursiva creando nuevas variables CPFloatVar con copia de dominios
`void print()`	Muestra la expresión de forma entendible invocandose recursivamente.
`void subscribe(...)`	Suscribe todas las vistas que se encuentren en las hojas del arbol para que sean manipuladas por el propagador.
`void cancel(...)`	Cancela la suscripción de las vistas del arból.
`void update(...)`	Actualiza las nuevas vistas copiando los dominios de las variables originales.
`bool assigned()`	Determina si todas las vistas del arbol que representa la expresión han sido asignadas.
`int countViews()`	Recorre el arbol contando las vistas econtradas en las hojas.
`void collectViews(ViewArray&)`	Recorre el árbol recogiendo todas las vistas encontradas y llenandolas en una ViewArray.
`INTERVAL evaluate()`	Evalúa en forma ascendente cada operación desde las hojas hacia la raiz resumiendo en cada paso los dominios totalizados.
`ExecStatus propagate(...)`	Poda cada dominio iniciando por la raiz hasta llegar a las hojas donde al final podará a las variables.

Clase Constraint *Constraint* es una clase que no pertenece a la familia de clases *Expression*, pero se comporta de forma similar a la clase *BinaryExpression* ya que también tiene dos expresiones asociadas, las cuales son relacionadas por igualdad. Esta clase también tiene implementados los métodos de tipo operador.

Clase HC4 Una vez dispuestos todos los recursos para construir y administrar un árbol con estas características, la clase *HC4* recibe como primer parámetro una referencia de la raíz del árbol, este primer nodo es un objeto de tipo *Constrain*, el cual a su vez tiene dos referencias que encadenan tanto por izquierda como por derecha a dos expresiones formadas recursivamente con objetos de la familia de *Expression*.

HC4 es un propagador que trabaja recursivamente el árbol recibido como referencia. Para propagar, ejecuta recursivamente a un método denominado *evaluate(...)* y una vez que este método ha llenado

Cuadro 10: Descripción de la clase *Expression*

`Expression()`	Constructor protegido para que no pueda ser instanciado por el usuario programador.
`~Expression()`	Destructor
`INTERVAL getInterval()`	Averigua el dominio resumido por el método evaluate() en cada nodo del árbol.
`Expression& sin()`	Crea una expresión unaria con la operación *seno* de la expresión.
`Expression& cos()`	Crea una expresión unaria con la operación *coseno* de la expresión.
`Expression& sqrt()`	Crea una expresión unaria con la operación $\sqrt{}$ de la expresión.
`Expression& operator+(...)`	Crea una expresion binaria con la operación $+$ entre dos expresiones o entre una expresión y una constante.
`Expression& operator-(...)`	Crea una expresion binaria con la operación $-$ entre dos expresiones o entre una expresión y una constante.
`Expression& operator*(...)`	Crea una expresion binaria con la operación $*$ entre dos expresiones o entre una expresión y una constante.
`Expression& operator/(...)`	Crea una expresion binaria con la operación $/$ entre dos expresiones o entre una expresión y una constante.
`Expression& operator^(...)`	Crea una expresion binaria con la operación de exponenciación ($\hat{}$) entre una expresión y una constante.
`Expression& operator==(...)`	Crea un objeto de tipo Constraint relacionando por igualdad dos expresiones o una expresión y una constante.

el árbol con resultados paraciales, ejecuta el método *propagate(...)* con el cual se realizan las podas en los dominios de las variables encadendas en las hojas del árbol, implementando de esta forma el algoritmo propuesto por Benamout [3]. El Cuadro 19 describe los miembros implementados para esta clase.

3.3.2. Propagador K3B

Este propagador esta programado siguiendo la técnica *3B-Consistency*, la cual se apoya en el propagador *HC4* para filtrar valores haciendo pruebas en los límites de cada uno de los intervalos de las variables relacionadas en la expresión.

Cuadro 11: Descripción de la clase *VarExpression* - Constructores y destructor

`VarExpression(CPFloatVar)`	Constructor que inicializa un objeto VarExpresión a partir de una variables CPFloatVar. Este constructor es utilizado para crear los nodos originales.
`VarExpression(CPFloatView)`	Constructor que inicializa un objeto VarExpresión utilizando otra vista. Este constructor es utilizado para clonar el arbol.
`~VarExpression()`	Destructor, en este nivel del árbol no se ejecuta nada.

Cuadro 12: Descripción de la clase *VarExpression* - Métodos Recursivos

`Expression& clone()`	Hace una copia de la vista manteniendo la referencia a la variable CPFloatVar.
`Expression& cloneToUpdate()`	Crea una nueva vista copiándole el dominio de la variable original.
`void print()`	Imprime el dominio de la variable CPFloatVar.
`void subscribe(...)`	Suscribe la vista asociada al nodo para que pueda ser trabajada en el espacio de busqueda.
`void cancel(...)`	Cancela la suscripción de la vistas.
`void update(...)`	Actualiza la nueva vista copiandole el dominio de la vista original.
`bool assigned()`	Determina si la variable ya fue solucionada.
`int countViews()`	Incrementa en uno (1) el contador de vistas.
`void collectViews(ViewArray&)`	Añade la vista al Arrego de Vistas.
`INTERVAL evaluate()`	Retorna el dominio de la variable CPFloatVar que pueda ser evaluada por los nodos superiores..
`ExecStatus propagate(...)`	Poda la variable CPFloatVar dependiendo del dominio relacionado con la expresión.

Este propagador se puede combinar con cualquier otro propagador que utilice la técnica *2B-Consistency*. En este proyecto, se ha limitado este apareamiento a trabajar *K3B* con *HC4*.

El poder de eliminados de valores que no cumplen con restricción puede ser más potente que el de *HC4*, sin embargo el propagador *K3B* internamente realiza un procedimiento más dispendioso al intentar podar que el realizado por *HC4*. La Figura 6 muestra la relación entre estas dos clases.

Cuadro 13: Descripción de la clase *ConstExpression* - Constructor y destructor

`ConstExpression(BoundType)`	Inicializa el objeto con un valor (double) del mismo tipo de los valores del dominio de las variables CPFloatVar.
`~ConstExpression()`	Destructor, en este nivel del árbol no se ejecuta nada.

Cuadro 14: Descripción de la clase *ConstExpression* - Métodos Recursivos

`Expression& clone()`	Hace una copia del valor constante.
`Expression& cloneToUpdate()`	Hace una copia del valor constante.
`void print()`	Imprime el valor de la constante (double).
`void subscribe(...)`	Método Vacio.
`void cancel(...)`	Método Vacio.
`void update(...)`	Método Vacio.
`bool assigned()`	Siempre retorna *true*.
`int countViews()`	No incrementa nada al contador de vistas
`void collectViews(ViewArray&)`	Método Vacio.
`INTERVAL evaluate()`	Devuelve el mismo valor como límites del Intervalo.
`ExecStatus propagate(...)`	Método Vacio.

Cuadro 15: Descripción de la clase *UnaryExpression* - Constructor y destructor

`UnaryExpression(Expression&,ExprType)`	Construye una nueva expresión operando a una sola expresión previemente creada, la operación que implementa es definida por el parámetro *ExprType*. Este método esta definido como privado para que no pueda ser instanciada directamente.
`~UnaryExpression()`	El método destructor invoca la destrucción de la expresion asociada por la operación.

3.4. Distribuidores

Dentro del módulo del sistema de clases, también se han diseñado 4 diferentes distribuidores, encargados de definir las políticas de selección de variables y valores al momento de realizar las búsquedas.

En la Figura 7 se distingue un nuevo espacio denominado “Branch”, creado para almacenar las

Cuadro 16: Descripción de la clase *UnaryExpression* - Métodos Recursivos

`Expression& clone()`	Invoca recursivamente al mismo método de la expresión asociada.
`Expression& cloneToUpdate()`	Invoca recursivamente al mismo método de la expresión asociada.
`void print()`	Imprime la operación definida en el objeto e invoca recursivamente al mismo método de la expresión asociada.
`void subscribe(...)`	Invoca recursivamente al mismo método de la expresión asociada.
`void cancel(...)`	Invoca recursivamente al mismo método de la expresión asociada.
`void update(...)`	Invoca recursivamente al mismo método de la expresión asociada.
`bool assigned()`	Invoca recursivamente al mismo método de la expresión asociada.
`int countViews()`	Invoca recursivamente al mismo método de la expresión asociada.
`void collectViews(ViewArray&)`	Invoca recursivamente al mismo método de la expresión asociada.
`INTERVAL evaluate()`	Invoca recursivamente al mismo método de la expresión asociada.
`ExecStatus propagate(...)`	Invoca recursivamente al mismo método de la expresión asociada.

Cuadro 17: Descripción de la clase *BinaryExpression* - Constructor y destructor

`BinaryExpression(Expression&,Expression&,ExprType)`
Inicializa una nueva expresión operando a dos expresiones previemente creadas, la operación que implementa es definida por el parámetro *ExprType*. Este método esta definido como privado para que no pueda ser instanciada directamente.
`~BinaryExpression()`
El método destructor invoca la destrucción de las dos expresiones asociadas por la operación.

nuevas clases (área de color gris). Los 4 distribuidores trabajan con 4 heurísticas diferentes de búsqueda.

Todos los distribuidores heredan sus características de la clase Brancher del sistema de clases de *Gecode*. También están asociadas con la clase CPFloatView por medio de la cual pueden consultar

Cuadro 18: Descripción de la clase *BinaryExpression* - Métodos Recursivos

`Expression& clone()`	Invoca recursivamente al mismo método de las dos expresiones asociadas.
`Expression& cloneToUpdate()`	Invoca recursivamente al mismo método de las dos expresiones asociadas.
`void print()`	Imprime la operación definida en el objeto e invoca recursivamente al mismo método de las dos expresiones asociadas.
`void subscribe(...)`	Invoca recursivamente al mismo método de las dos expresiones asociadas.
`void cancel(...)`	Invoca recursivamente al mismo método de las dos expresiones asociadas.
`void update(...)`	Invoca recursivamente al mismo método de las dos expresiones asociadas.
`bool assigned()`	Invoca recursivamente al mismo método de las dos expresiones asociadas.
`int countViews()`	Invoca recursivamente al mismo método de las dos expresiones asociadas.
`void collectViews(ViewArray&)`	Invoca recursivamente al mismo método de las dos expresiones asociadas.
`INTERVAL evaluate()`	Invoca recursivamente al mismo método de las dos expresiones asociadas.
`ExecStatus propagate(...)`	Invoca recursivamente al mismo método de las dos expresiones asociadas.

valores de variables y modificarlos.

Las características de estos distribuidores se detallan en las siguientes subsecciones.

3.4.1. NoneMin

Esta clase está diseñada e implementada para hacer la distribución usando una sola variable. Cuando el motor de búsqueda requiere dividir el dominio para hacer las pruebas necesarias, NoneMin divide el dominio justo por la mitad. A esta heurística también se le denomina SPLIT.

Esta clase provee 2 servicios especiales importantes para definir la política de distribución, el Cuadro 21 los relaciona sin tener en cuenta los métodos adicionales obligatorios del sistema.

El usuario programador podrá utilizar este distribuidor por medio de la función branch(...), ubicada en el espacio CPFloat, codificada por fuera de la clase NoneMin.

Cuadro 19: Miembros para propagador *HC4*

`cnst_ //constraint`	Variable que referencia a la raiz del árbol credo por la expresión entregada al Propagador
`HC4(Space&,Constraint&)`	Inicia el propagador recibiendo como parametros el espacio de las variables y la raiz del árbol que tiene la expresión.
`HC4(Space&,bool,HC4&)`	Constructor de Copia del Propagador.
`Propagator* copy(Space&,bool)`	Métodos para clonar el propagador.
`ExecStatus propagate(Space&,ModEventDelta&)`	Este método es el encargado de podar los dominios de las variables invocando a los métodos evalutate(...) y propagate(...) recursivos en el árbol.
`ExecStatus post(Space&,Constraint&)`	Desde aquí se autoinstancia la clase HC4. Este método es invocado por la funcion externa *hc4(...)* utilizada por el usuario programador.

3.4.2. FirstFail

Esta clase está implementada para distribuir sobre un arreglo de variables. Cuando el motor de búsqueda requiere dividir el espacio para hacer las pruebas necesarias, FirstFail selecciona la variable que tenga el dominio más reducido y luego divide su dominio justo por la mitad.

FirstFail provee 2 servicios para aplicar la distribución, estos servicios son listados en el Cuadro 22.

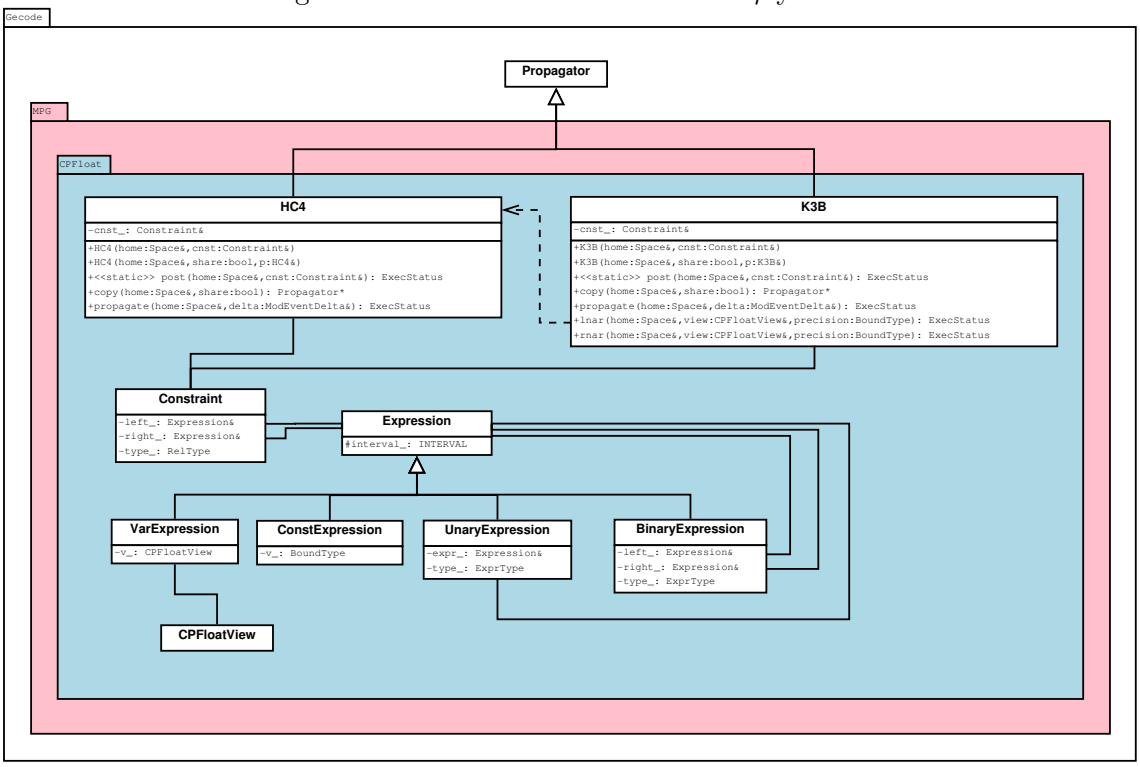
El usuario programador podrá utilizar este distribuidor por medio de la función firstfail(...), la cual al igual que branch(), esta por fuera de la clase FirstFail.

3.4.3. Naive

Al igual que FirstFail, Naive también trabaja sobre un arreglo de variables contando con los mismos servicios presentados en el Cuadro 22. Solo se diferencian por la heurística utilizada para la búsqueda. Naive selecciona la variable a dividir, en forma ordenada de izquierda a derecha.

El usuario programador podrá utilizar este distribuidor por medio de la función naive(...), codificado por fuera de la clase.

Figura 6: Relación de clases entre *HC4* y *K3B*



3.4.4. Rand

La clase Rand tiene una estructura similar a la de Naive e igualmente trabaja sobre un arreglo de variables contando con los mismos servicios listados en el Cuadro 22. En su heurística, Rand selecciona la variable a dividir, en forma aleatoria.

El usuario programador podrá utilizar este distribuidor por medio de la función randselection(...) codificado por fuera de la clase.

Cuadro 20: Miembros para propagador K3B

`cnst_ //constraint`	Variable que referencia a la raiz del árbol credo por la expresión entregada al Propagador
`K3B(Space&,Constraint&)`	Inicia el propagador recibiendo como parametros el espacio de las variables y la raiz del árbol que tiene la expresión.
`K3B(Space&,bool,K3B&)`	Constructor de Copia del Propagador.
`Propagator* copy(Space&,bool)`	Métodos para clonar el propagador.
`ExecStatus propagate(Space&,ModEventDelta&)`	Este método es el encargado de podar los dominios de las variables ejecutando sobre cada una de las variables un filtro en los extremos. El Filtro en cada extremo de cada variable es el mismo utilizado por *HC4*.
`ExecStatus post(Space&,Constraint&)`	Desde aquí se autoinstancia la clase K3B. Este método es invocado por la funcion externa *k3b(...)* utilizada por el usuario programador.

Cuadro 21: Métodos de la clase NoneMin

`choice(...) //choice a split point`	Selecciona el punto de división del dominio y escoge alternadamente el lado izquierdo o el derecho.
`commit(...) //update domain of variable`	Actualiza los dominios después de haber seleccionado un bloque de prueba con el método choice.

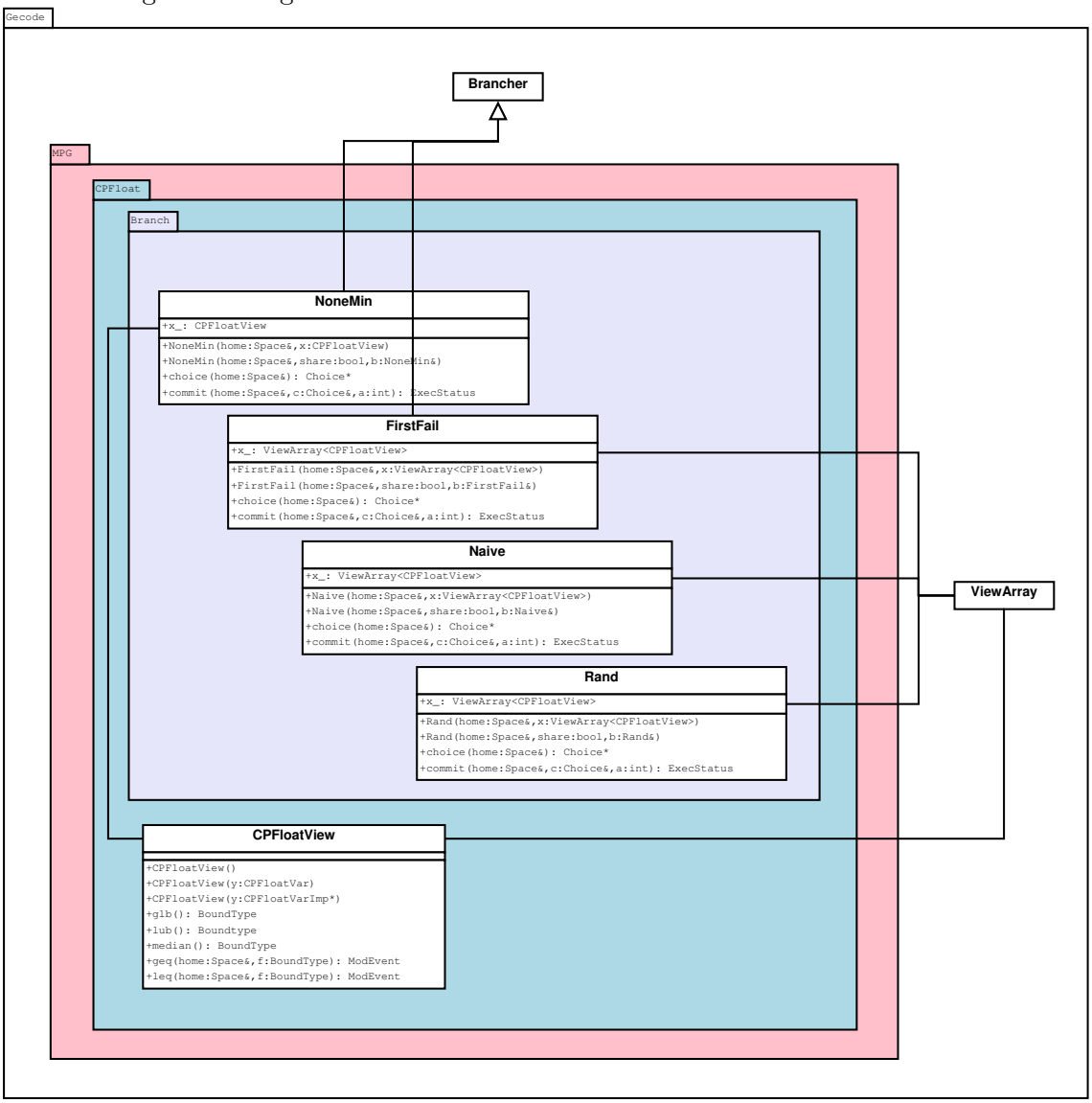
4. Capítulo 4

4.1. Pruebas

4.1.1. Benchmarks con HC4

Para confrontar los resultados se tomarán 5 ejemplos ya implementados en codigo *Mozart* bajo el proyecto *XRI*, razón por la cual se podrá fácimente comparar los resultados. Estos algoritmos están listados con sus particularidades en el proyecto COCONUT (Continuous Constraints - Updating the Technology) [20]. Los ejemplos a tomar serán: Bronstein, Freudenstein, Cyclo, Collins, Chemistry,

Figura 7: Diagrama del Sistema de Distribuidores de *CPFloat-Gecode*



Bellido y Broyden.

Bronstein Este problema requiere resolver el valor de 3 variables de acuerdo con el siguiente sistema de ecuaciones:

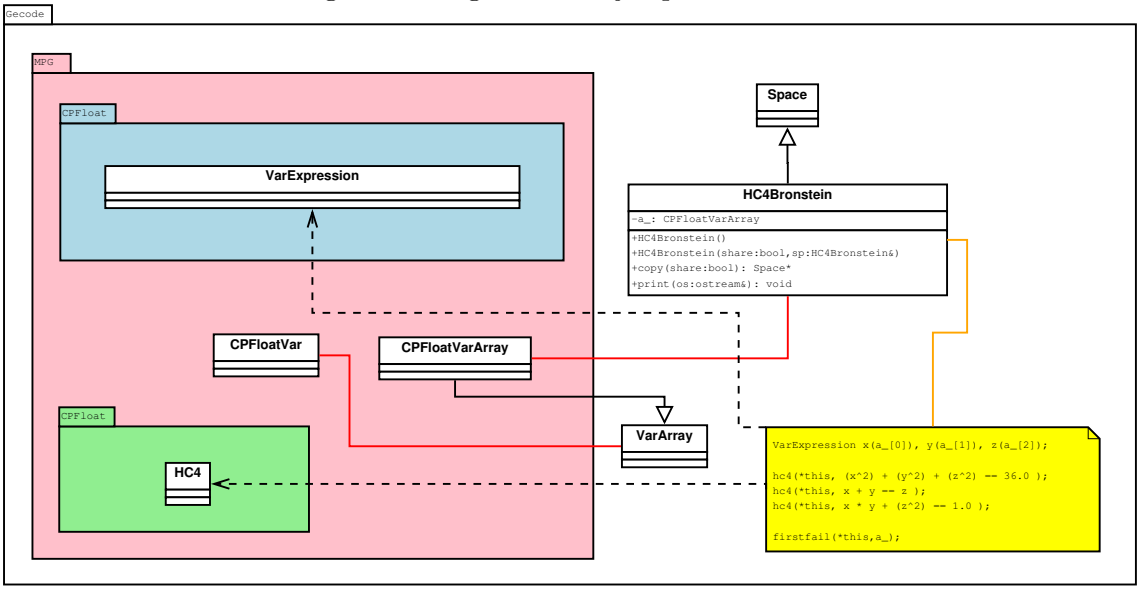
$x^2 + y^2 + z^2 = 36$
$x + y = z$
$x * y + z^2 = 1$

Cuadro 22: Métodos de la clase FirstFail

`choice(...) //choice a variable and a split point`	Selecciona la variable a dividir y escoge un punto de división del dominio, posteriormente elige alternadamente el lado izquierdo o el derecho..
`commit(...) //update domain of variable`	Actualiza los dominios después de haber seleccionado un bloque de prueba con el método choice.

El ejercicio fue resuelto utilizando el módulo *CPFloat* agregado a *Gecode* según como se muestra en la Figura 8

Figura 8: Diagrama del ejemplo Bronstein



La implementación de este ejercicio es relativamente sencilla, se crea una clase nueva heredada de la clase *Space* sobrecargando algunos métodos como los mostrados en la Figura 8. La transcripción del CSP también es intuitiva, en la Figura 8 está presentada en la caja de mensaje de color amarillo.

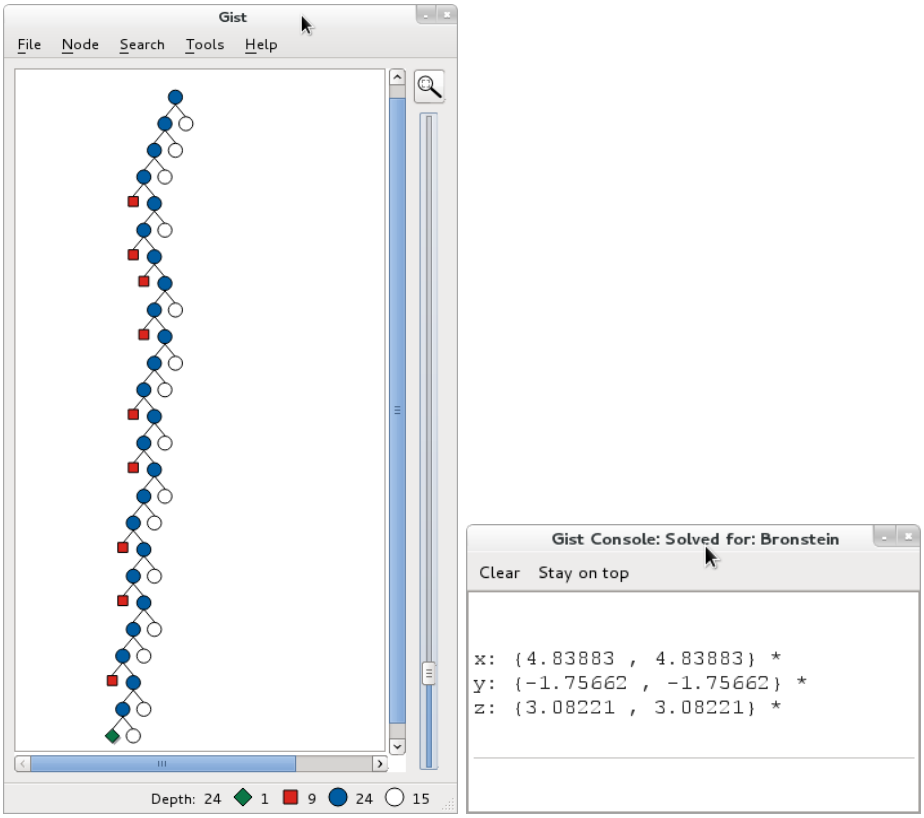
Aunque en el diagrama de la Figura 8 no se presenta el módulo *Gist* (Necesario para graficar el arbol de busqueda), el resultado será presentado utilizando la rutina *dfs* (Depth First Search) de dicho módulo.

Una vez ejecutado el programa, se obtuvo como solución la siguiente: $x = 4{,}83883$, $y = -1{,}75662$, $z = 3{,}08221$.

En la Figura 9 se muestra el árbol de búsqueda con el cual se encontró la primera solución del sistema de ecuaciones de Bronstein, alcanzando una profundidad de 24 pasos. La solución es presentada

por los intervalos encerrados entre llaves "{ }". El sigo asterisco "∗" impreso al final del intervalo, significa que la variable ya fue asignada, es decir, que el tamaño del intervalo es inferior a la precisión deseada (en este ejemplo se utilzó la precisión por defecto que corresponde a 10^{-9}).

Figura 9: Resultado de Bronstein solucionado con *Gecode*



Según el portal web del proyecto COCONUT, la mejor solución encontrada es: $x(1) = -1{,}7566229888$, $x(2) = 4{,}8388299903$, $x(3) = 3{,}0822070015$.

Como parte del proyecto, se comparó el resultado también con el obtenido en mozart con el módulo *XRI*. La Figura 10 muestra la ejecución del mismo problema solucionado con *Mozart*.

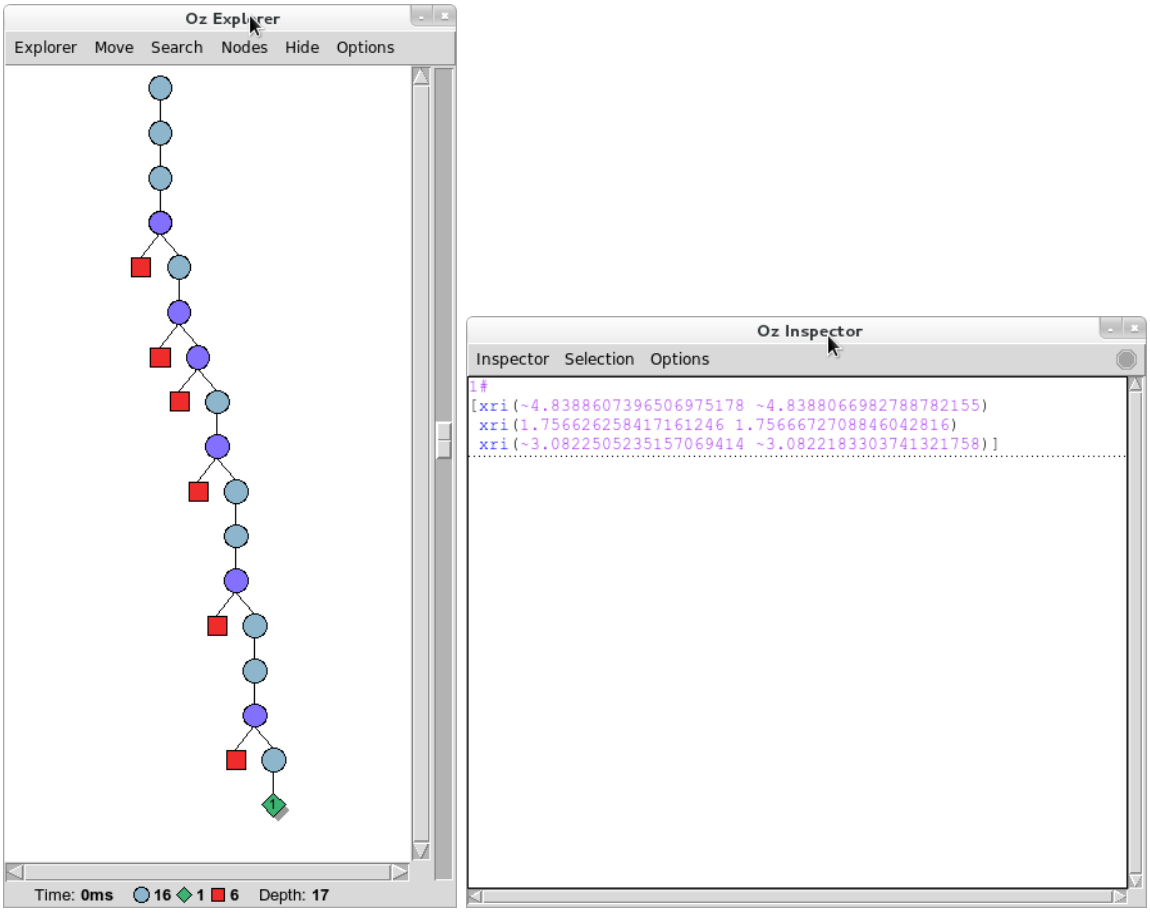
Los resultados varían en el signo debido a que la primera solución encontrada por *Gecode* es una simetría de la solución encontrada por *Mozart*, pero de igual forma ambas son solución.

Esta simetría es ocasionada por la diferencia en codificación de los distribuidores de cada uno de los módulos de restricciones (*CPFloat-Gecode* y *XRI-Mozart*).

Freudenstein De acuerdo al siguiente sistema de ecuaciones, se requiere encontrar el valor de 2 variables $x1$ y $x2$.

$x1 - 29 + (x2(x2 + 1) - 14) * x2 = 0$

Figura 10: Resultado de Bronstein solucionado con *Mozart*



$x1 - 13 + (x2(5 - x2) - 2) * x2 = 0$

La codificación de este ejercicio se presenta en la Figura 11, siguiendo el mismo esquema del ejemplo anterior. A diferencia de Bronstein, para este ejemplo se ha utilizado como heurística de distribución *randselection* que busca en forma aleatoria una variable para hacer la ramificación. Fueron probados otros modelos de distribución pero en todos, la búsqueda fue extensa alcanzando una profundidad muy lejana sin encontrar solución.

La Figura 12 muestra el resultado de la busqueda utilizando el módulo *CPFloat* de *Gecode*. Al final de 58 pasos econtŕo como solución: $x1 = 5$, $x2 = 4$.

También se codificó el mismo ejemplo con el módulo *XRI* de mozart con el cual se obtuvo el mismo resultado pero utilizando más pasos, la Figura 13 presenta la ejecución en mozart con una profundidad de 75 pasos.

Figura 11: Diagrama del ejemplo Freudenstein

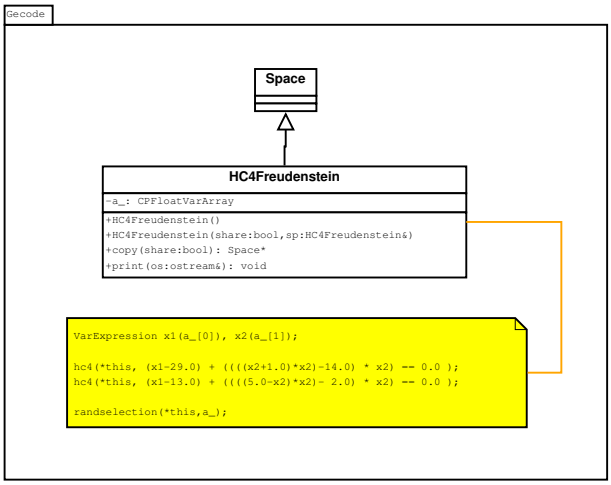
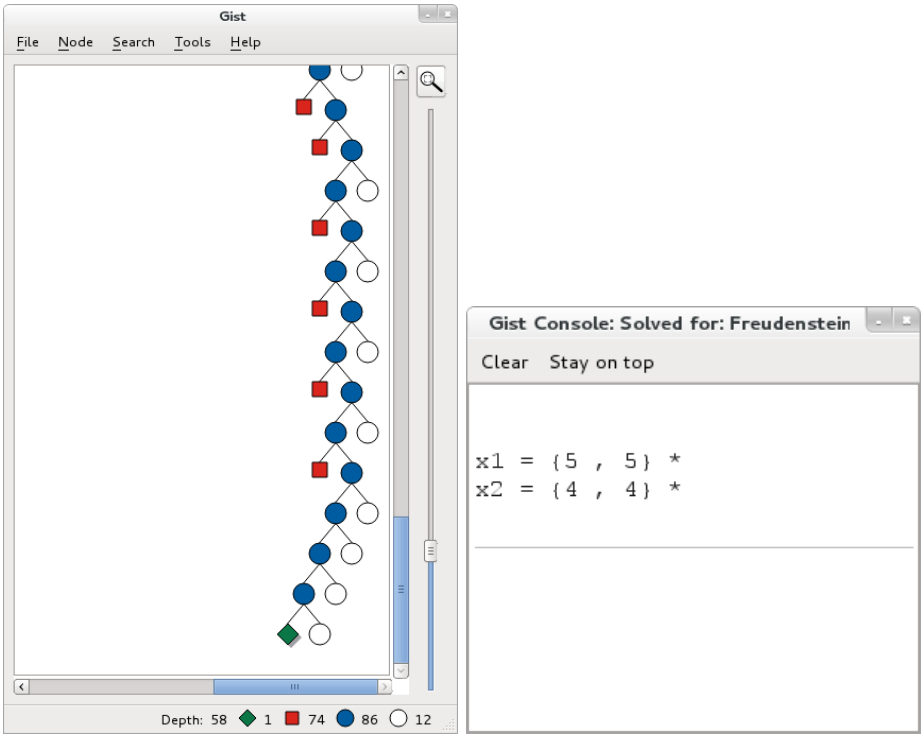
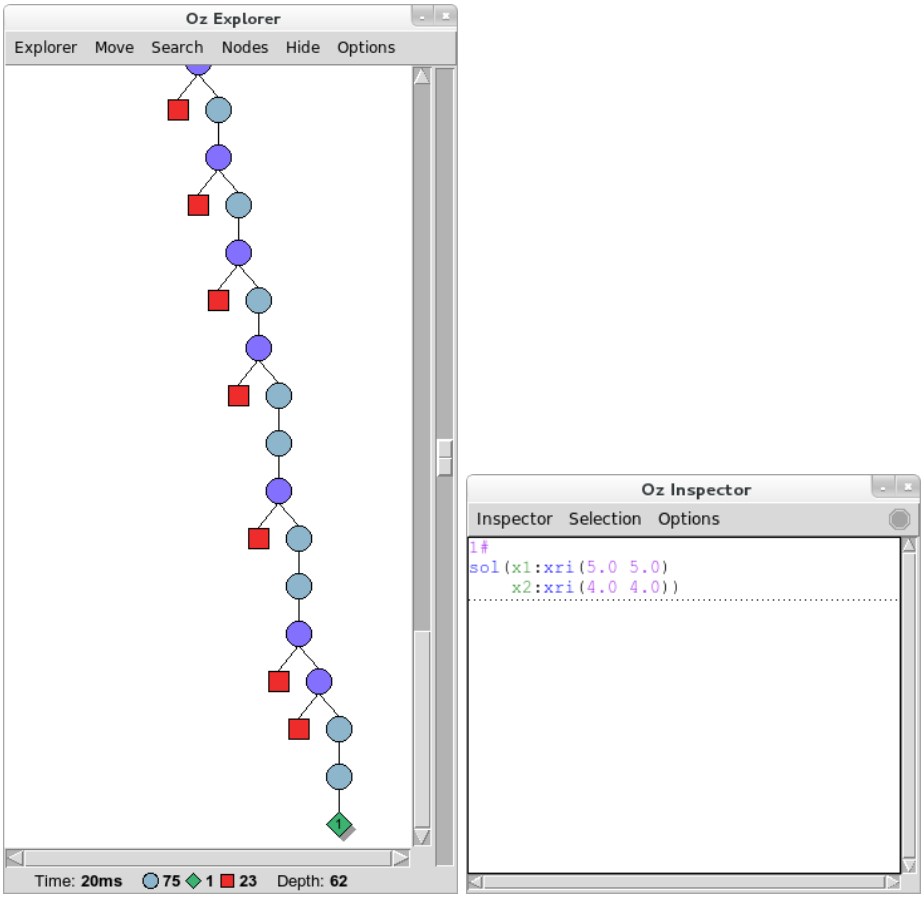


Figura 12: Resultado de Freudenstein solucionado con *Gecode*



Cyclo Versión *XRI* Esta versión de Cyclo fue tomada de los ejemplos realizados en el proyecto *XRI* de mozart, el cual contiene un sistema de ecuaciones diferente al presentado en el proyecto COCONUT. En el siguiente aparte se presentará los resultados de la versión COCONUT.

Figura 13: Resultado de Freudenstein solucionado con *Mozart*



El ejemplo fue codificado según como se muestra en la Figura 14, de aquí en adelante los diagramas de los ejemplos serán resumidos, se asume que las relaciones con las clases paquete CPFloat, son las mismas.

Para encontrar la solución fue requerido utilizar *randselection* en el distribuidor, ya que otras heurísticas no pudieron econtrar una solución en corto tiempo.

La Figura 15 presenta el resultado de la ejecución de este problema en *Gecode*, encontrando una de las soluciones después de una profundidad de 55 pasos.

Debido a que la selección de las variables en la distribución de la búsqueda es aleatoria, las soluciones que se encuentran pueden ser diferentes. En esta instancia los valores encontrados fueron: $x = 5{,}19909$, $y = 4{,}10348$, $z = 0{,}321729$.

La ejecución del mismo problema en *Mozart* encontró otra solución despues de hacer una búsqueda con una profundiad de 254 pasos. El resultado de la ejecución se presenta en la Figura 16.

Figura 14: Diagrama del ejemplo Cyclo Versión *XRI*

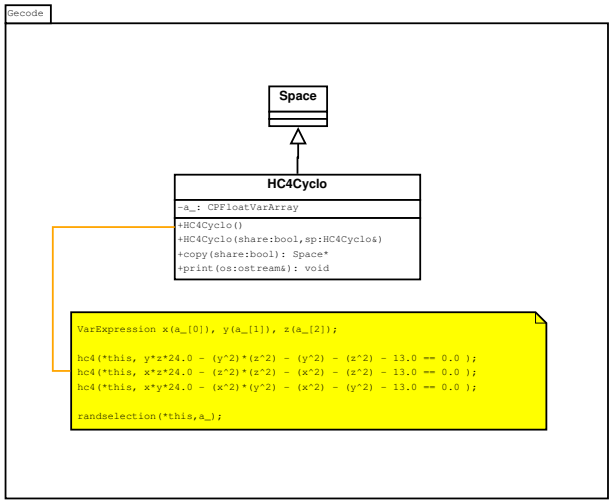


Figura 15: Resultado de Cyclo Version *XRI* solucionado con *Gecode*

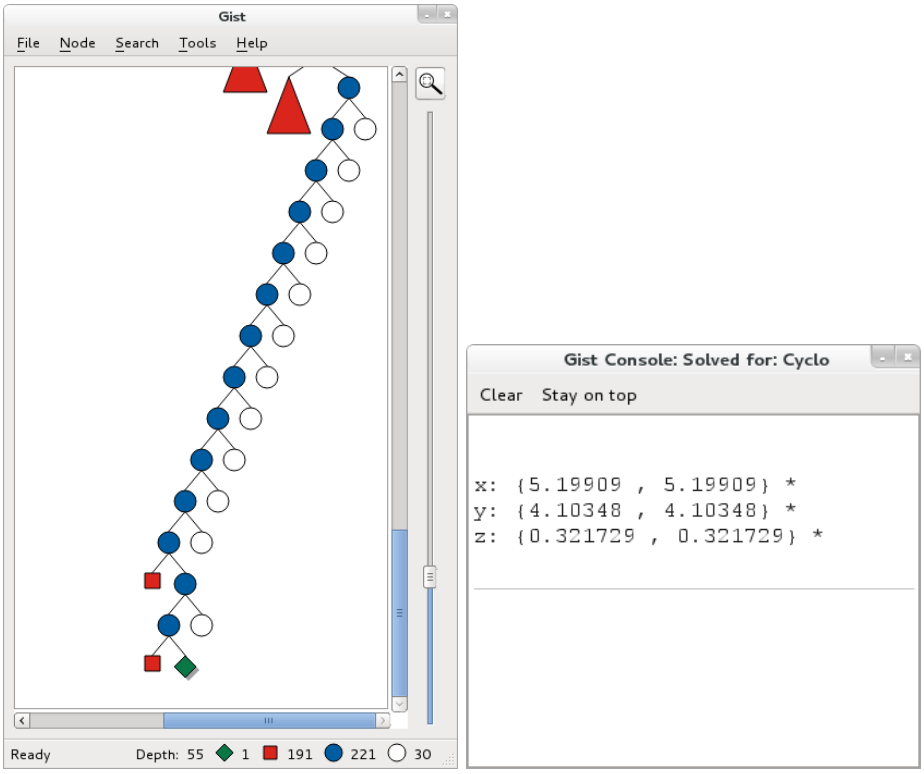
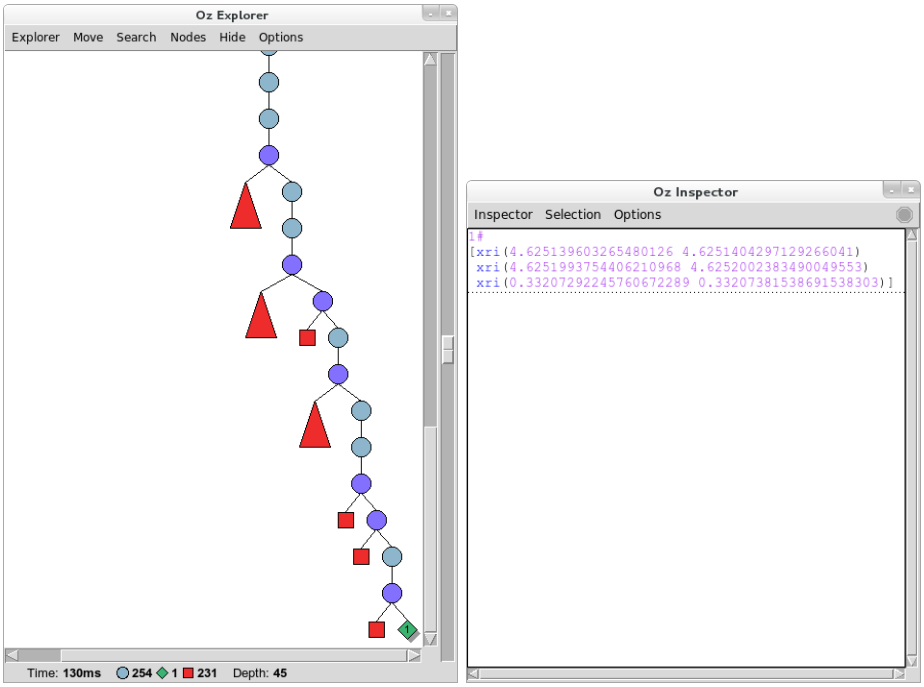


Figura 16: Resultado de Cyclo Version *XRI* solucionado con *Mozart*



Cyclo Versión COCONUT Según el sitio oficial del proyecto COCONUT [20], el sistema de ecuaciones a resolver es el siguiente:

$$959 * t2^2 + 774 * t3^2 + 1389 * t2 * t3 + 1313 * t2^2 * t3^2 = 310$$
$$755 * t3^2 + 917 * t1^2 + 1451 * t3 * t1 + 269 * t3^2 * t1^2 = 365$$
$$837 * t1^2 + 838 * t2^2 + 1655 * t1 * t2 + 1352 * t1^2 * t2^2 = 413$$

La codificación en *Gecode* es transcrita en forma similar, el diagrama de la Figura 17 muestra la versión de *Cyclo* según COCONUT.

En la misma pagina web de COCONUT se presenta con mejor solución la siguiente:

$x(1) = 0{,}373753$
$x(2) = 0{,}314479$
$x(3) = 0{,}302819$

Y la ejecución en *Gecode* buscando la solución para este problema encontró la misma solución: $x = 0{,}373753$, $y = 0{,}314479$, $z = 0{,}302819$. La Figura 18 presenta el árbol de búsqueda y la solución después de la ejecución.

Figura 17: Diagrama del ejemplo Cyclo Versión COCONUT

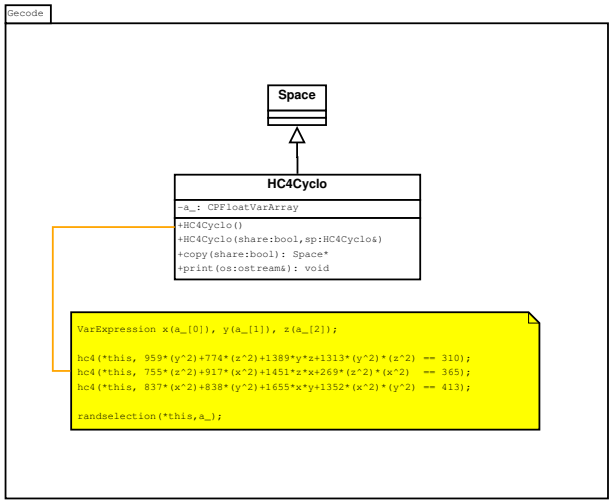
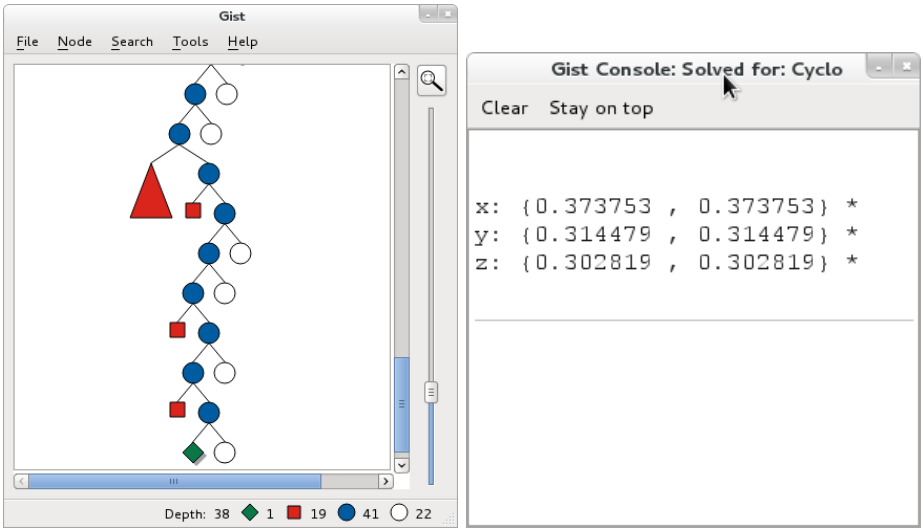


Figura 18: Resultado de Cyclo Version COCONUT solucionado con *Gecode*

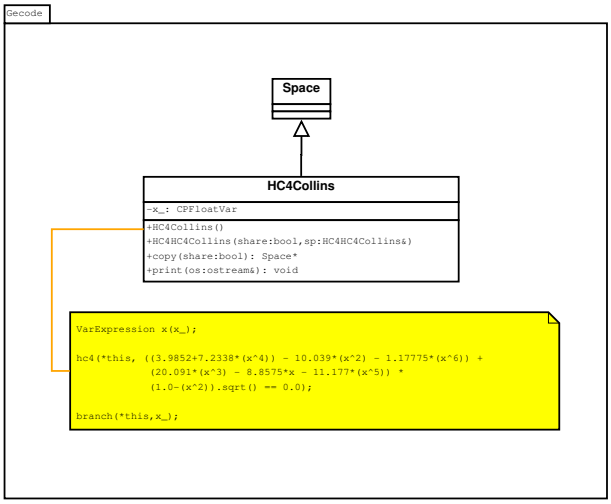


Collins Este problema fue tomado de los ejemplos de *XRI* en mozart, donde la ecuación solo requiere solucionar el valor de una sola variable:

$$(3{,}9852 + 7{,}2338x^4 - 10{,}039x^2 - 1{,}17775x^6) + (20{,}091x^3 - 8{,}8575x - 11{,}177x^5) * \sqrt{1 - x^2} = 0$$

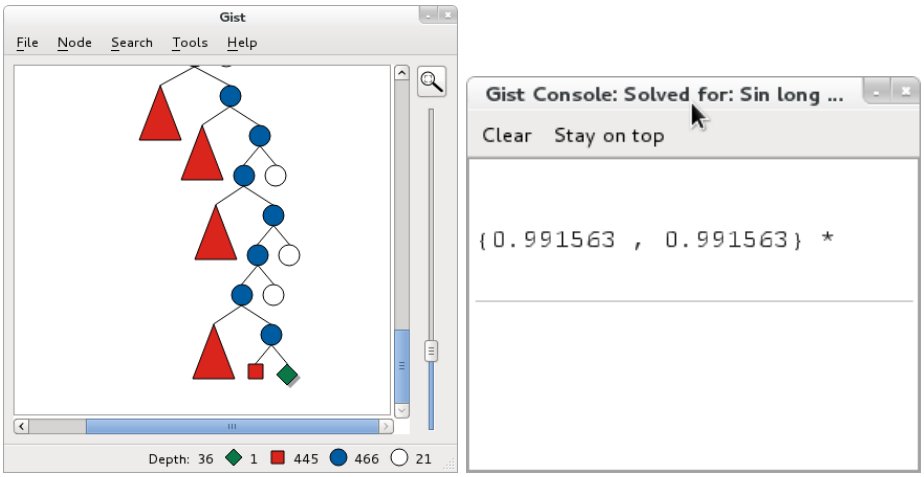
La expresión de esta ecuación fue escrita en *Gecode* tal como se muestra en la Figura 19, donde existe una única variable x a diferencia de los anteriores ejemplos que utilizaban un arreglo de variables. Igualmente se utilizó como heurística de distribución el procedimiento *branch*, el cual siempre divide el intervalo de la variable por la mitad.

Figura 19: Diagrama del ejemplo Collins



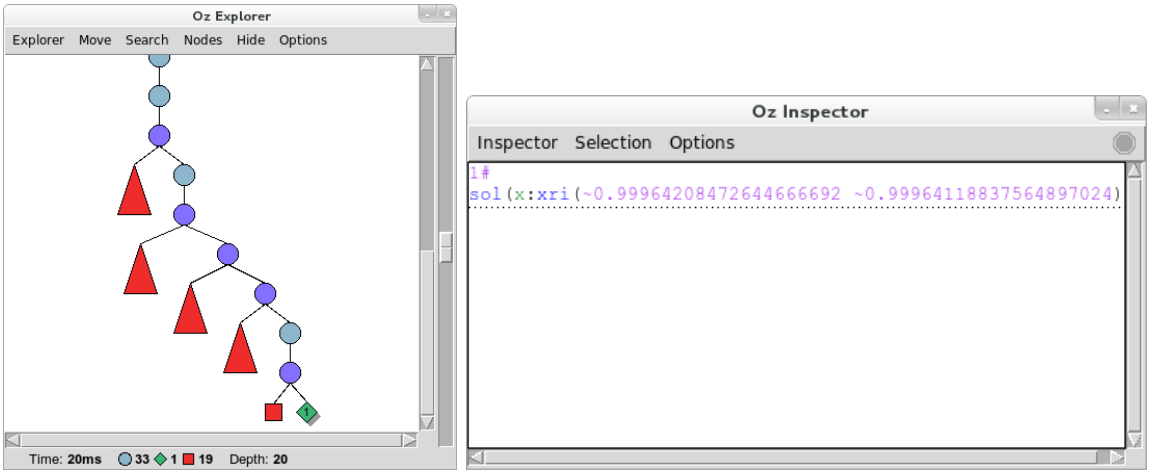
El resultado de la ejecución en *Gecode* se muestra en la Figura 20 encontrando la solución después de 36 pasos de profundiad. $x = 0{,}991563$.

Figura 20: Resultado de Collins solucionado con *Gecode*



La misma ejecución en *Mozart* encontró la solución en 33 pasos, este resultado se puede ver en la Figura 21.

Figura 21: Resultado de Collins solucionado con *Mozart*



4.1.2. Benchmarks con K3B

Chemistry Este problema fue tomado de la página del proyecto COCONUT y requiere encontrar el valor de 5 variables usando los valores de las constantes $R, R5, R6, R7, R8, R9$ y $R10$ dentro del siguiente sistema de ecuaciones:

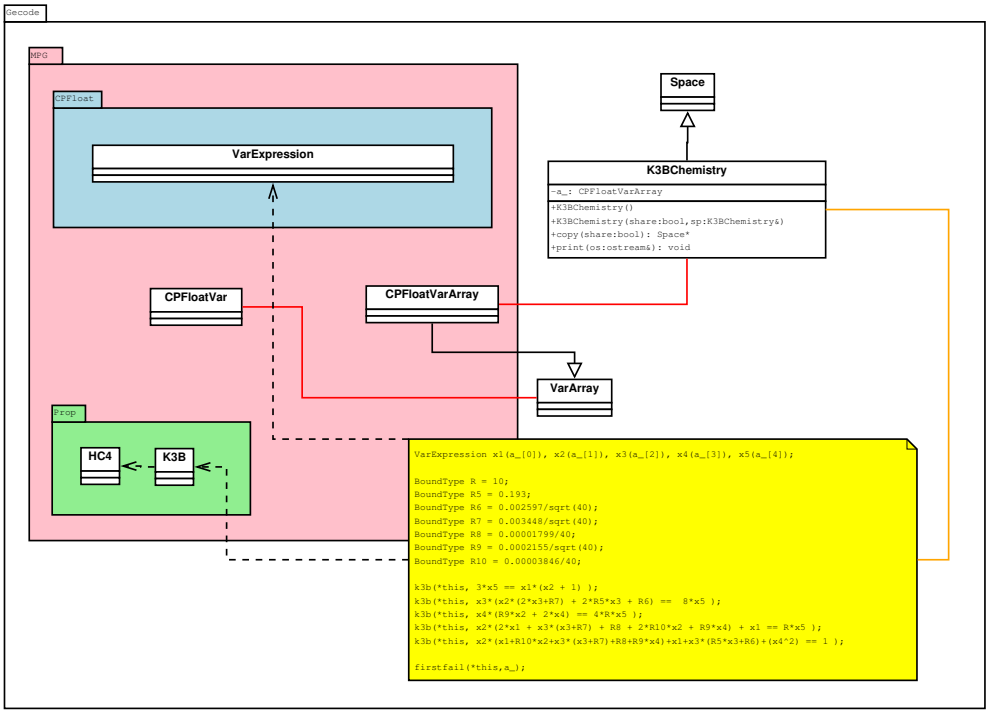
$R = 10;$
$R5 = 0{,}193$
$R6 = 0{,}002597/sqrt(40)$
$R7 = 0{,}003448/sqrt(40)$
$R8 = 0{,}00001799/40$
$R9 = 0{,}0002155/sqrt(40)$
$R10 = 0{,}00003846/40$

$3 * x5 = x1 * (x2 + 1)$
$x3 * (x2 * (2 * x3 + R7) + 2 * R5 * x3 + R6) = 8 * x5$
$x4 * (R9 * x2 + 2 * x4) = 4 * R * x5$
$x2 * (2 * x1 + x3 * (x3 + R7) + R8 + 2 * R10 * x2 + R9 * x4) + x1 = R * x5$
$x2 * (x1 + R10 * x2 + x3 * (x3 + R7) + R8 + R9 * x4) + x1 + x3 * (R5 * x3 + R6) + x4^2 = 1$

El sistema de ecuaciones propuesto anteriormente se codificó en *Gecode* usando el propagador K3b, de forma similar a como se codificaron los ejemplos que utilizaron HC4. Este ejercicio está diseñado tal como se muestra en la figura 22

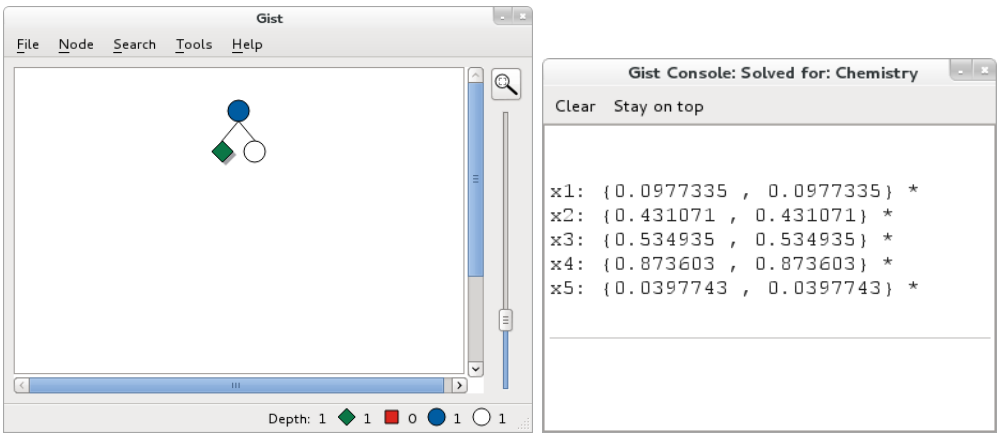
La primera solución fue encontrada por *Gecode* a un solo paso de profundidad así como se muestra en la Figura 23. Los valores para las variables son: $x1 = 0{,}0977335$, $x2 = 0{,}431071$, $x3 = 0{,}534935$, $x4 = 0{,}873603$ y $x5 = 0{,}0397743$.

Figura 22: Diagrama del ejemplo Chemistry



K3B es un propagador que asegura mayor reducción de dominios, el desempeño de este ejercicio será evaluado en el Capítulo 5.

Figura 23: Resultado de Chemistry solucionado con *Gecode*



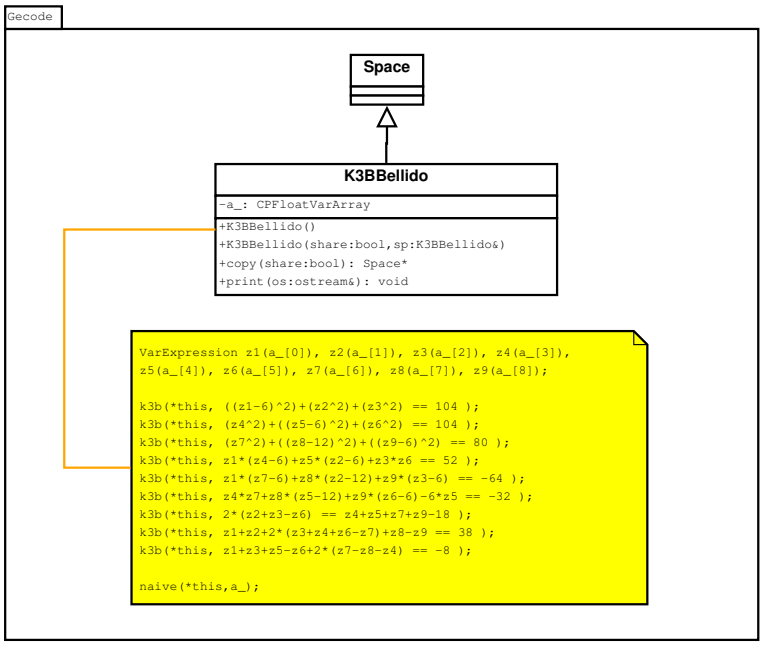
Bellido Este problema también fue tomado del sitio web oficial del proyecto COCONUT, en el cual se requiere encontrar el valor de 9 variables de acuerdo al siguiente sistema de ecuaciones:

$(z1-6)^2+z2^2+z3^2=104$
$z4^2+(z5-6)^2+z6^2=104$
$z7^2+(z8-12)^2+(z9-6)^2=80$
$z1*(z4-6)+z5*(z2-6)+z3*z6=52$
$z1*(z7-6)+z8*(z2-12)+z9*(z3-6)=-64$
$z4*z7+z8*(z5-12)+z9*(z6-6)-6*z5=-32$
$2*(z2+z3-z6)=z4+z5+z7+z9-18$
$z1+z2+2*(z3+z4+z6-z7)+z8-z9=38$
$z1+z3+z5-z6+2*(z7-z8-z4)=-8$

Este problema fue solucionado utilizando el propagador K3B ya que el propagador HC4 no logró hacerlo en un periodo de tiempo razonable.

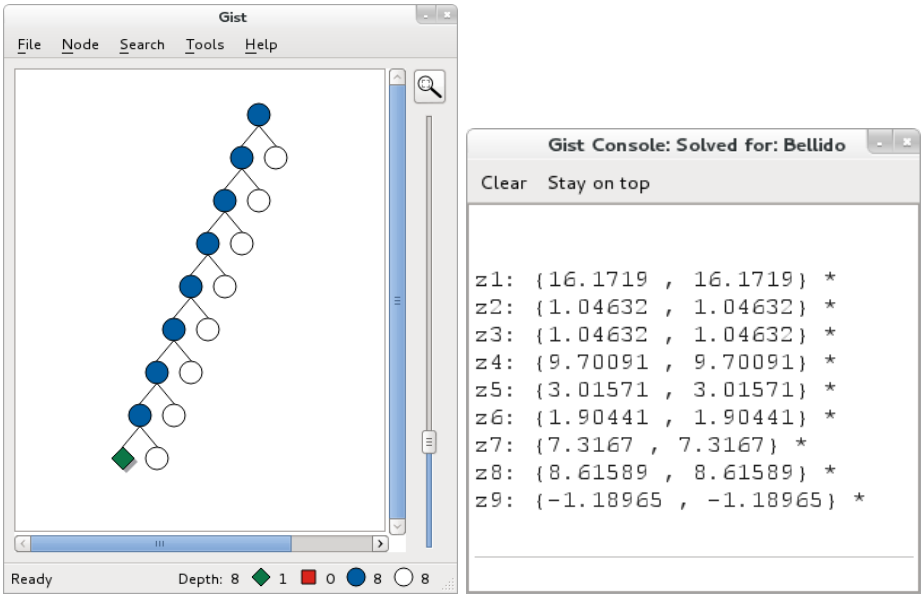
La Figura 24 muestra la implementación codificada en *Gecode*. La ejecución en la que se obtiene una primera solución logró hacerlo en 8 pasos: $z1 = 16{,}1719$, $z2 = 1{,}04632$, $z3 = 1{,}04632$, $z4 = 9{,}70091$, $z5 = 3{,}01571$, $z6 = 1{,}90441$, $z7 = 7{,}3167$, $z8 = 8{,}61589$ y $z9 = -1{,}18965$.

Figura 24: Diagrama del ejemplo Bellido



La Figura 25 presenta el resultado de la ejecución de este ejercicio utilizando como heurística de distribución un método simple y ordenado (*naive*), el cual no siempre es eficiente.

Figura 25: Resultado de Bellido solucionado con *Gecode*



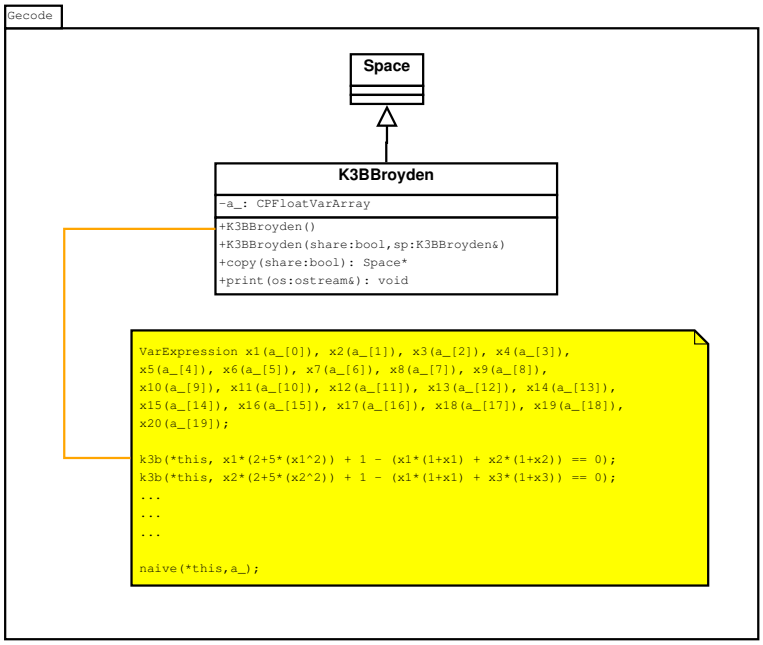
Broyden Desde el sitio web oficial del proyecto COCONUT, se tomó el sistema de ecuaciones para Broyden20, con el cual se busca resolver el valor de 20 variables. Las ecuaciones siguientes fueron codificadas.

$x1 * (2 + 5 * x1^2) + 1 - (x1 * (1 + x1) + x2 * (1 + x2)) = 0$
$x2 * (2 + 5 * x2^2) + 1 - (x1 * (1 + x1) + x3 * (1 + x3)) = 0$
$x3 * (2 + 5 * x3^2) + 1 - (x1 * (1 + x1) + x2 * (1 + x2) + x4 * (1 + x4)) = 0$
$x4 * (2 + 5 * x4^2) + 1 - (x1 * (1 + x1) + x2 * (1 + x2) + x3 * (1 + x3) + x5 * (1 + x5)) = 0$
$x5 * (2 + 5 * x5^2) + 1 - (x1 * (1 + x1) + x2 * (1 + x2) + x3 * (1 + x3) + x4 * (1 + x4) + x6 * (1 + x6)) = 0$
$x6 * (2 + 5 * x6^2) + 1 - (x1 * (1 + x1) + x2 * (1 + x2) + x3 * (1 + x3) + x4 * (1 + x4) + x5 * (1 + x5) + x7 * (1 + x7)) = 0$
$x7 * (2 + 5 * x7^2) + 1 - (x2 * (1 + x2) + x3 * (1 + x3) + x4 * (1 + x4) + x5 * (1 + x5) + x6 * (1 + x6) + x8 * (1 + x8)) = 0$
$x8 * (2 + 5 * x8^2) + 1 - (x3 * (1 + x3) + x4 * (1 + x4) + x5 * (1 + x5) + x6 * (1 + x6) + x7 * (1 + x7) + x9 * (1 + x9)) = 0$
$x9 * (2 + 5 * x9^2) + 1 - (x4 * (1 + x4) + x5 * (1 + x5) + x6 * (1 + x6) + x7 * (1 + x7) + x8 * (1 + x8) + x10 * (1 + x10)) = 0$
$x10 * (2 + 5 * x10^2) + 1 - (x5 * (1 + x5) + x6 * (1 + x6) + x7 * (1 + x7) + x8 * (1 + x8) + x9 * (1 + x9) + x11 * (1 + x11)) = 0$
$x11 * (2 + 5 * x11^2) + 1 - (x6 * (1 + x6) + x7 * (1 + x7) + x8 * (1 + x8) + x9 * (1 + x9) + x10 * (1 + x10) + x12 * (1 + x12)) = 0$
$x12 * (2 + 5 * x12^2) + 1 - (x7 * (1 + x7) + x8 * (1 + x8) + x9 * (1 + x9) + x10 * (1 + x10) + x11 * (1 + x11) + x13 * (1 + x13)) = 0$
$x13 * (2 + 5 * x13^2) + 1 - (x8 * (1 + x8) + x9 * (1 + x9) + x10 * (1 + x10) + x11 * (1 + x11) + x12 * (1 + x12) + x14 * (1 + x14)) = 0$
$x14 * (2 + 5 * x14^2) + 1 - (x9 * (1 + x9) + x10 * (1 + x10) + x11 * (1 + x11) + x12 * (1 + x12) + x13 * (1 + x13) + x15 * (1 + x15)) = 0$

$x15 * (2 + 5 * x15^2) + 1 - (x10 * (1 + x10) + x11 * (1 + x11) + x12 * (1 + x12) + x13 * (1 + x13) + x14 * (1 + x14) + x16 * (1 + x16)) = 0$
$x16 * (2 + 5 * x16^2) + 1 - (x11 * (1 + x11) + x12 * (1 + x12) + x13 * (1 + x13) + x14 * (1 + x14) + x15 * (1 + x15) + x17 * (1 + x17)) = 0$
$x17 * (2 + 5 * x17^2) + 1 - (x12 * (1 + x12) + x13 * (1 + x13) + x14 * (1 + x14) + x15 * (1 + x15) + x16 * (1 + x16) + x18 * (1 + x18)) = 0$
$x18 * (2 + 5 * x18^2) + 1 - (x13 * (1 + x13) + x14 * (1 + x14) + x15 * (1 + x15) + x16 * (1 + x16) + x17 * (1 + x17) + x19 * (1 + x19)) = 0$
$x19 * (2 + 5 * x19^2) + 1 - (x14 * (1 + x14) + x15 * (1 + x15) + x16 * (1 + x16) + x17 * (1 + x17) + x18 * (1 + x18) + x20 * (1 + x20)) = 0$
$x20*(2+5*x20^2)+1-(x15*(1+x15)+x16*(1+x16)+x17*(1+x17)+x18*(1+x18)+x19*(1+x19)) = 0$

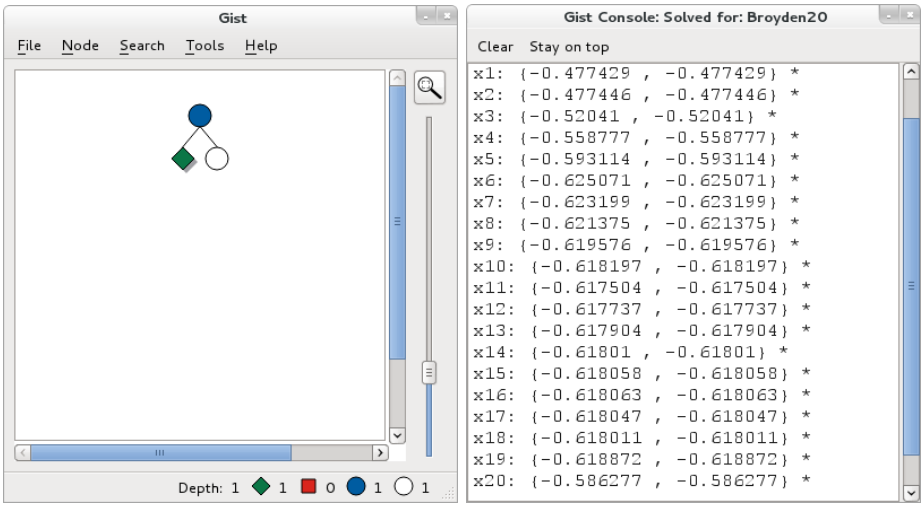
Gracias al propagador K3B, fue posible la implementación del problema en *Gecode* de acuerdo a lo planteado la Figura 26.

Figura 26: Diagrama del ejemplo Broyden



La Figura 27 presenta el resultado de la ejecución de este ejercicio en el cual, se logra encontrar una solución a un paso de profunidad en la búsqueda.

Figura 27: Resultado de Broyden solucionado con *Gecode*



5. Capítulo 5

5.1. Resultados Obtenidos

Una vez completada la implementación del Sistema de Restricciones para Dominios Continuos para *Gecode*, se logró resolver problemas que requerían trabajar con valores reales. En el Capítulo 4 se presentaron 7 ejemplos solucionados como respuesta a problemas presentados como Benchmarks en el sitio oficial del proyecto COCONUT [20]. Los resultados obtenidos después de este ejercicio, más las pruebas hechas por fuera de lo informado en este documento, corroboran el alcance de los objetivos planteados en la propuesta inicial.

Todos los ejercicios planteados a excepción de Freudenstein, fueron transcritos de una forma natural, es decir, se consultaron directamente en la web del proyecto COCONUT copiándose las ecuaciones (Ctr+C) y se pegaron en el código de *Gecode* (Ctr+V), realizándoles pequeños ajustes.

En el caso de Freudenstein se tradujo el problema que estaba previamente escrito en *Mozart* con la sintaxis del módulo *XRI*.

Con esta facilidad de codificación, se cumple uno de los objetivos que consistía en hacer que los propagadores puedan ser utilizados de la forma más natural posible.

Tal como se pudo observar, todas las pruebas usaron pocos pasos para encontrar una solución: Bronstein 24, Freudenstein 58, Cyclo Versión *XRI* 55, Cyclo Version COCONUT 38, Collins 36, Chemistry 1, Bellido 8 y Broyden 1.

Para comprobar los resultados se compararon los valores con dos fuentes diferentes: La primera fuente fue el conjunto de ejercicios resueltos con el sistema de restricciones para dominios continuos de mozart (*XRI*) y la segunda fue la información suministrada por el sitio Web oficial del proyecto

COCONUT. Por este motivo, el problema *Cyclo* tiene dos versiones.

En el Capitulo 4, los problemas fueron presentados con la primera de las soluciones encontradas en cuyo caso no siempre coincidió con las mejores respuestas listadas en el portal web de COCONUT, esto se debe a que los algoritmos del distribuidor y las heurísticas utilizadas, son diferentes.

Teniendo en cuenta que es necesario verificar de alguna forma que las respuestas publicadas en los benchmarks también se pueden obtener con el sistema de restricciones de *Gecode* (*CPFloat-Gecode*), a continuación se realizan algunos análisis especiales de cada uno de los benchmarks.

Bronstein La solución encontrada por *CPFloat-Gecode* fue: $x = 4{,}83883, y = -1{,}75662, z = 3{,}08221$. Estos resultados coincidieron con los establecidos como los mejores resultados en el proyecto COCONUT: $x(1) = -1{,}7566229888, x(2) = 4{,}8388299903, x(3) = 3{,}0822070015$. En este resultado también se confrontó la solución encontrada con el módulo *XRI* de *Mozart*: $-4{,}838860...$, $1{,}756626...$, $-3{,}082250....$ Estos últimos están cercanos a los mejores resultados pero invertidos en signo debido a un proceso de simetría generado por la diferencia de algoritmos en el distribuidor que hace que la primera solución encontrada sea otra.

Freudenstein Es un problema que no fue tomado del proyecto COCONUT, sino de los ejercicios resueltos en *Mozart*. La solución encontrada por *CPFloat-Gecode* fue: $x1 = 5{,}0$, $x2 = 4{,}0$; resultado que coincide con el contrado por el módulo *XRI* de *Mozart*: $x1 = 5{,}0$, $x2 = 4{,}0$.

Cyclo Versión *XRI* Este problema se lo encontró definido en *Mozart* diferente a cómo está planteado en el proyecto COCONUT, por tal motivo, se probó el módulo *CPFloat-Gecode* con las dos versiones. Para la versión *XRI*, *CPFloat-Gecode* encontró como primera solución la siguiente: $x = 5{,}19909$, $y = 4{,}10348$, $z = 0{,}321729$. Teniendo en cuenta que la heurística de distribución consistía en seleccionar una variable aleatóriamente, no fue posible hacer que la ejecución con mozart coincidiera encontrando la misma respuesta. $4{,}6251396...$, $4{,}6251993...$, $0{,}332072....$ Sin embargo, cualquiera de las dos soluciones es válida.

Cyclo Versión COCONUT En este problema *CPFloat-Gecode* encontró como primera solución $x = 0{,}373753$, $y = 0{,}314479$, $z = 0{,}302819$, la cual coincide con la mejor solución registrada en el proyecto COCONUT: $x(1) = 0{,}373753$, $x(2) = 0{,}314479$, $x(3) = 0{,}302819$.

Collins La primera solución encontrada por *CPFloat-Gecode* fue $x = 0{,}991563$, no es la misma primera solución encontrada por *XRI-Mozart* la cual fue $x = 0{,}99964208472644666692$. Sin embargo, despues de hacer el cálculo con los valores entregados, la solución encontrada por *CPFloat-Gecode* es más precisa que la de *XRI de Mozart*, tal como se muestra en el Cuadro 23.

Chemistry Este problema fue resuelto utilizando el propagador K3B de *CPFloag-Gecode* y la primera solución encontrada fue: $x1 = 0{,}0977335$, $x2 = 0{,}431071$, $x3 = 0{,}534935$, $x4 = 0{,}873603$ y $x5 = 0{,}0397743$. Debido a la implementación del algoritmo, esta primera solución no coincide

Cuadro 23: Comparación de Soluciones para Collins

Solución con *CPFloat-Gecode*	Solución con *XRI-Mozart*
$-1{,}63035e^{-7} \simeq 0$	$0{,}00316523 \simeq 0$

con la mejor solución propuesta por COCONUT: $x1 = 0{,}0031141$, $x2 = 34{,}5979$, $x3 = 0{,}0650418$, $x4 = 0{,}859378$ y $x5 = 0{,}0369518$. Con el ánimo de verificar si *CPFloat-Gecode* es capaz de encontrar dicha solución, se hizo un acercamiento en el árbol de búsqueda, asignando las dos primeras variables con dos propagadores como los que se muestran en el Cuadro 24 para $x1$ y $x2$; obteniendo iguales resultados en las otras variables: $x3 = 0{,}0650418$, $x4 = 0{,}859378$ y $x5 = 0{,}0369518$.

Cuadro 24: Restricciones adicionales para Chemistry

```
k3b(*this, x1 == 0.0031141);
k3b(*this, x2 == 34.5979);
```

Bellido Este sistema de ecuaciones que no pudo ser resuelto con el propagador *HC4*, encontró como solución la siguiente: $z1 = 16{,}1719$, $z2 = 1{,}04632$, $z3 = 1{,}04632$, $z4 = 9{,}70091$, $z5 = 3{,}01571$, $z6 = 1{,}90441$, $z7 = 7{,}3167$, $z8 = 8{,}61589$ y $z9 = -1{,}18965$. En el sitio oficial del proyecto COCONUT, la mejor solución es $z1 = 9{,}39167$, $z2 = 9{,}24763$, $z3 = 2{,}64156$, $z4 = 7{,}96267$, $z5 = 8{,}00825$, $z6 = 2{,}89272$, $z7 = 6{,}95595$, $z8 = 14{,}3723$ y $z = 11{,}1139$. De la misma forma como se realizó el testeo con Chemistry, se buscó con *CPFloat-Gecode* la misma respueta acercándonos al resultado con la restricciones presentadas en el Cuadro 25.

Cuadro 25: Restricciones adicionales para Bellido

```
hc4(*this, z1 == 9.3916661681 );
hc4(*this, z2 == 9.2476345419 );
hc4(*this, z3 == 2.6415631705 );
hc4(*this, z4 == 7.9626675077 );
```

Broyden La solución encontrada para las 20 variables por *K3B* en *CPFloat-Gecode* fue: $x1 = -0{,}477429$, $x2 = -0{,}477446$, $x3 = -0{,}52041$, $x4 = -0{,}558777$, $x5 = -0{,}593114$, $x6 = -0{,}625071$, $x7 = -0{,}623199$, $x8 = -0{,}621375$, $x9 = -0{,}619576$, $x10 = -0{,}618197$, $x11 = -0{,}617504$, $x12 = -0{,}617737$, $x13 = -0{,}617904$, $x14 = -0{,}61801$, $x15 = -0{,}618058$, $x16 = -0{,}618063$, $x17 = -0{,}618047$, $x18 = -0{,}618011$, $x19 = -0{,}618872$ y $x20 = -0{,}586277$. Esta respuesta coincide con la mejor solución publicada en el proyecto COCONUT.

6. Capítulo 6

6.1. Conclusiones

Una vez completado el trabajo propuesto al inicio de esta investigación, el módulo creado *CPFloat-Gecode* cumple con los objetivos propuestos, en tanto ahora con *Gecode*, es posible codificar un CSP cuyas variables almacenan valores continuos. Se agregaron 33 clases, necesarias para cumplir con el doble objetivo del proyecto: por un lado, la posibilidad de trabajar con restricciones sobre variables continuas, y por el otro lado, mantener la estructura filosófica de extensibilidad de *Gecode*.

Las 33 clases están repartidas asi: 10 Clases del sistema base, 11 para propagadores matemáticos básicos, 8 para propagadores avanzados (HC4 y K3B) y 4 para distribuidores.

La primera parte del proyecto que contempló el sistema base de *CPFloat-Gecode*, fue una primera etapa clave para entender con claridad el manejo apropiado de valores reales fundamentándose en la teoría de la Aritmética Intervalar [10]. Adicionalmente, fue el momento para aprender a manipular correctamente el sistema de redondeo para números flotantes [5], gracias a los cuales se pudo representar valores reales encerrados en intervalos limitados por números flotantes.

La segunda etapa que contempló la implementación de los propagadores avanzados *HC4* y K3B, fue una fase que requirió de bastante estudio de los algoritmos propuestos por Benamout [2][3] y las teorías de Collavizza [4] y Lhomme [11].

En la etapa de Pruebas expuesta en el Capitulo 4, se puede apreciar la funcionalidad del sistema de restricciones de *CPFloat-Gecode*, en el cual se documentaron 8 ejercicios seleccionados cuyos resultados pudieron ser confrontados. Para este efecto se seleccionaron dos fuentes a saber: La documentación presentada en el proyecto COCONUT [20], donde no solo se exponen los problemas, sino también las mejores respuestas; y la otra fuente son los ejercicios resueltos también en el sistema de restricciones para dominios continuos de *XRI-Mozart*.

Una vez confrontados los resultados se encontró que el desempeño de *CPFloat-Gecode* es adecuado, y los tiempos de respuesta se asimilan a los obtenidos por *XRI-Mozart*.

Cabe anotar que dentro del proyecto se le dió una especial importancia a la interface con el usuario programador que usará *CPFloat-Gecode*, por tal motivo, se implementaron métodos especiales que sobrecargan los operadores matemáticos utilizados en este proyecto (+,-,*, ...) de tal forma que la codificación de restricciones se hiciese de la forma más natural posible.

6.2. Trabajos Futuros

Como es de esperarse este proyecto inició un camino en el ambiente de restricciones sobre dominios continuos sobre *Gecode* el cual no contaba hasta el momento con un módulo que soportara este trabajo.

Por tal motivo, se deja como trabajos futuros algunas características adicionales que a juicio del investigador, deberían continuarse en un próximo proyecto ligado a este primero.

En cuanto al **Sistema Base**, es necesario hacer una revisión general y acondicionar las rutinas necesarias para que el sistema de redondeo de números flotantes no solo trabaje para arquitecturas de 32bits y 64bits, sino tambien para otras arquitecturas.

Los **Propagadores Básicos** propuestos en este proyecto deberían ser complementados de tal forma que el conjunto de propagadores de *CPFloat-Gecode*, permita implementar más tipos de restricciones,como por ejemplo, relaciones de desigualdades ($<$, $>$, ...).

Respecto a los **Propagadores Avanzados**, se podría implementar otros algoritmos como *Box Consistency* que de acuerdo a la teoría revisada, puede en algunos casos lograr una poda más fuerte al momento de hacer la búsqueda de soluciones.

En general, se podrían realizar depuraciones algorítmicas que al ser realizadas minuciosamente, pueden mejorar el performance del sistema de *CPFloag-Gecode*.

Referencias

[1] K. Apt. Principles of Contraint Programming. Cambridge. 2003.

[2] F. Benhamou. Interval constraint logic programming. Constraint Programming : Basics and Trends,LNCS 910 . 1995

[3] F. Benhamou, F. Goualard, L. Granvilliers and J.-F. Puget. Revising hull and box consistency. in Proceedings of ICLP '99. 1999

[4] H. Collavizza, F. Delobel, and M. Rueher. Comparing Partial Consistencies. Reliable Computing, 1999.

[5] IEEE Standard for Binary Floating-Point Arithmetic. IEEE Std 754-1985 (Reaffirmed 1990). IEEE Computer Society.

[6] G. Caicedo et la.. Loss reduction in Distribution Networks using concurrent constraint programming. 8th International Conference on Probabilistic Methods Applied to Power Systems. (PMAPS-2004)

[7] A. Cuyt et al.. A remarkable example of catastrophic cancellation unraveled. Computing, 66(2001). 2001.

[8] J. Diaz, C. Rueda, G. Gutierrez and C. Olarte. CRE2: A CP Application for Reconfiguring a Power Distribution Network for Power Losses Reduction. Alemania Lecture Notes In Computer Science. Springer v.3258 fasc.1 p.813 - 813 ,2004

[9] J. Diaz, C. Lozano, I. Romero. Solving the Short Run Economic Dispatch Problem Using Concurrent Constraint Programming. International Federation For Information Processing, Volume 218. 2006

[10] T. J. Hickey, Q. Ju, and M. H. Van Emden. Interval Arithmetic: from Principles to Implementation. J. ACM, volume 48, issue 5, (September 2001).

[11] O. Lhomme, Consistency Techniques for Numeric CSPs. in Proc. IJCAI93, Chambery, (France), pp. 232-238, August 1993.

[12] R. Moore. Interval Analysis. Prentice Hall. 1966

[13] http://en.wikipedia.org/wiki/Floating_point. Consultada en Octubre de 2009.

[14] http://www.mozart-oz.org/. Consultada en Octubre de 2009.

[15] http://www.gecode.org/. Consultada en Octubre de 2009.

[16] https://github.com/yjaradin/mozart2/wiki/Mozart-Virtual-Machine-Implementation. Consultada en Noviembre e 2011.

[17] http://www.boost.org/doc/libs/1_40_0/libs/numeric/interval/doc/interval.htm. Consultada en Octubre de 2009.

[18] http://cic.puj.edu.co/wiki/doku.php?id=grupos:avispa:geoz. Consultada en Octubre de 2009.

[19] http://home.gna.org/xrilpoz/. Consultada en Octubre de 2009.

[20] http://www.mat.univie.ac.at/~neum/glopt/coconut/Benchmark/Benchmark.html. Consultada en Enero 2010.